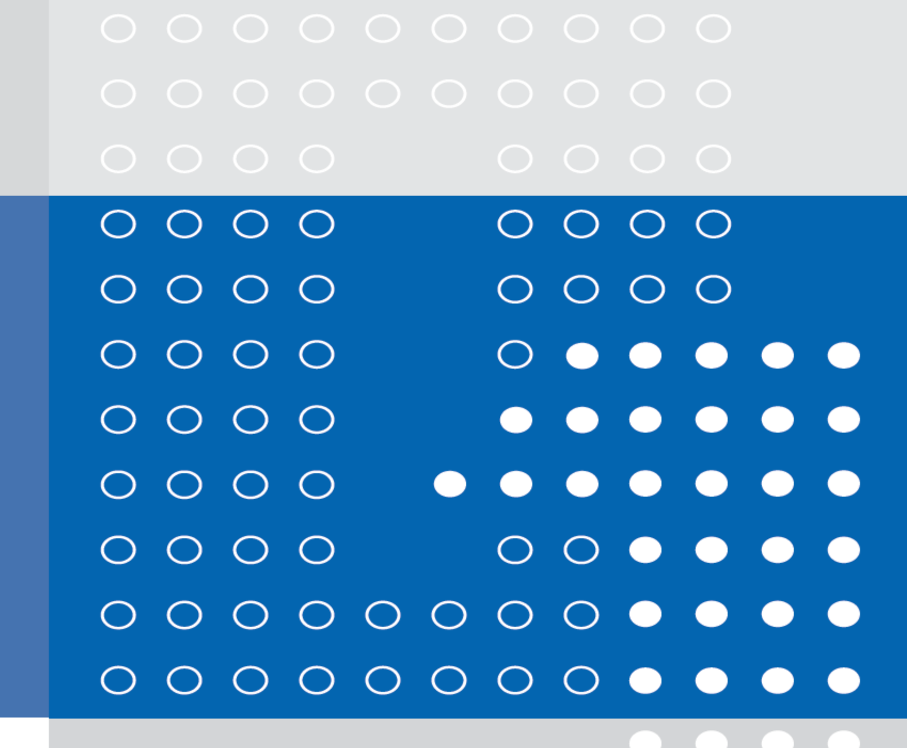
计算机系列教材

# Python程序设计基础



董付国　编著

清华大学出版社

计算机系列教材

# Python 程序设计基础

董付国　编著

清华大学出版社
北　　京

## 内 容 简 介

全书共 9 章，主要内容组织如下：第 1 章介绍 Python 基本知识与概念；第 2 章讲解列表、元组、字典、集合等序列的常用方法和基本操作；第 3 章讲解 Python 选择结构、for 循环与 while 循环、break 与 continue 语句；第 4 章讲解字符串编码格式，字符串格式化、替换、分割、连接等基本操作方法，正则表达式语法、正则表达式对象、子模式与 match 对象，以及正则表达式模块 re 的应用；第 5 章讲解函数的定义与使用，关键参数、默认值参数、长度可变参数、变量作用域以及 lambda 表达式；第 6 章讲解类的定义、类成员与实例成员、私有成员与公有成员、特殊方法与运算符重载；第 7 章讲解文件操作基本知识，文本文件内容读取与写入，二进制文件操作与对象序列化，文件复制、移动、重命名、MD5 值计算、压缩与解压缩等文件级操作以及目录操作有关知识；第 8 章讲解 Python 异常类层次结构，不同形式的异常处理结构，以及如何调试 Python 程序；第 9 章讲解如何使用 wxPython 进行 GUI 编程，主要包括窗体、按钮、文本框、单选钮、复选框等控件以及各种对话框的运用。

本书对 Python 内部工作原理进行了一定程度的剖析，对 Python 2.x 和 Python 3.x 之间的区别进行了深入对比和分析，并适当介绍 Python 程序优化和安全编程的有关知识，可以满足不同层次读者的需要。



**图书在版编目（CIP）数据**

Python 程序设计基础/董付国编著. --北京：清华大学出版社，2015
计算机系列教材
ISBN 978-7-302-41058-4

Ⅰ. ①P…　Ⅱ. ①董…　Ⅲ. ①软件工具－程序设计－高等学校－教材　Ⅳ. ①TP311.56

中国版本图书馆 CIP 数据核字(2015)第 173346 号

**责任编辑**：白立军　李　晔
**封面设计**：常雪影
**责任校对**：焦丽丽
**责任印制**：王静怡

**出版发行**：清华大学出版社
　**网　　址**：http://www.tup.com.cn，http://www.wqbook.com
　**地　　址**：北京清华大学学研大厦 A 座　　**邮　　编**：100084
　**社 总 机**：010-62770175　　**邮　　购**：010-62786544
　**投稿与读者服务**：010-62776969，c-service@tup.tsinghua.edu.cn
　**质量反馈**：010-62772015，zhiliang@tup.tsinghua.edu.cn
　**课件下载**：http://www.tup.com.cn,010-62795954
**印 装 者**：北京密云胶印厂
**经　　销**：全国新华书店
**开　　本**：185mm×260mm　　**印　　张**：16　　**字　　数**：372 千字
**版　　次**：2015 年 8 月第 1 版　　**印　　次**：2015 年 8 月第1次印刷
**印　　数**：1～2000
**定　　价**：29.00 元

---

产品编号：065428-01

# 前 言

Python 由 Guido van Rossum 于 1989 年底研制，第一个公开发行版本发行于 1991 年。Python 推出不久就迅速得到了各行业人士的青睐，经过二十多年的发展，已经渗透到计算机科学与技术、统计分析、移动终端开发、科学计算可视化、逆向工程与软件分析、图形图像处理、人工智能、游戏设计与策划、网站开发等几乎所有专业和领域。目前，Python 已经成为卡耐基-梅隆大学、麻省理工学院、加州大学伯克利分校、哈佛大学等国外很多大学计算机专业或非计算机专业的程序设计入门教学语言，国内也有不少学校的多个专业陆续开设了 Python 程序设计课程。Python 语言连续多年在 TIOBE 网站的编程语言排行榜上排名前十位，并于 2011 年 1 月被 TIOBE 网站评为 2010 年度语言。在 2014 年 12 月 IEEE Spectrum 推出的编程语言排行榜中，Python 更是取得了第 5 位的好名次。

Python 是一门免费、开源的跨平台高级动态编程语言，支持命令式编程、函数式编程，完全支持面向对象程序设计，语法简洁清晰，并且拥有大量功能丰富而强大的标准库和扩展库以及众多狂热的支持者，可以帮助各领域的科研人员或策划师甚至管理人员快速实现和验证自己的思路与创意。Python 用户可以把主要精力放在业务逻辑的设计与实现上，而不用过多考虑语言本身的细节，开发效率非常高，其精妙之处令人击节赞叹。

Python 是一门快乐的语言，学习和使用 Python 也是一个快乐的过程。与 C 语言系列和 Java 等语言相比，Python 更加容易学习和使用，但这并不意味着可以非常轻松地掌握 Python。熟练掌握和运用 Python 仍需要通过大量的练习来锻炼自己的思维和熟悉 Python 编程模式，同时还需要经常关注 Python 社区优秀的代码以及各种扩展库的动态。当然，如果能够适当了解 Python 及其扩展库的内部工作原理，对于编写正确而优雅的 Python 程序也是有很大帮助的。

Python 是一门优雅的语言。Python 语法简洁清晰，并且提供了大量的内置对象和内置函数，编程模式非常符合人类的思维方法和习惯。在有些编程语言中需要编写大量代码才能实现的功能，在 Python 中仅需要调用内置函数或内置对象的方法即可实现。如果有其他程序设计语言的基础，那么在学习和使用 Python 的时候，一定不要把其他语言的编程习惯和风格带到 Python 中来，因为这不仅可能会使得代码变得非常冗长、烦琐，还可能会严重影响代码的效率。应该尽量尝试从最自然、最简洁的角度出发去思考和解决问题，这样才能写出更加优雅、更加 Pythonic 的代码。

## 本书内容组织

对于 Python 程序员来说,能够熟练运用各种扩展库毫无疑问是非常重要的,使用优秀、成熟的扩展库可以帮助我们快速实现自己的业务逻辑和创意。但是也必须清楚地认识到,Python 语言基础知识和基本数据结构的熟练掌握是理解和运用其他扩展库的必备条件之一。因此,本书把重点和主要篇幅放在 Python 编程基础知识的介绍上,通过大量案例介绍 Python 在实际开发中的应用,关于不同应用领域的扩展库可以参考附录 B,并结合自己的专业领域查阅相关文档。全书共 9 章,主要内容组织如下:

第 1 章　基础知识。介绍如何选择 Python 版本,Python 对象模型,数字、字符串等基本数据类型,运算符与表达式,内置函数,基本输入输出,Python 程序文件名,扩展库的管理与使用,Python 代码编写规范,等等。

第 2 章　Python 序列。讲解序列常用方法和基本操作,成员测试运算符,切片操作,列表基本操作与常用方法,列表推导式,元组与生成器推导式,序列解包,字典、集合基本操作和常用方法,以及如何使用 Python 基本数据类型实现栈、二叉树、有向图等复杂数据结构。

第 3 章　选择与循环。讲解 Python 选择结构,for 循环与 while 循环,带有 else 子句的循环结构,break 与 continue 语句,选择结构与循环结构的综合运用。

第 4 章　字符串与正则表达式。讲解字符串编码格式,字符串格式化、替换、分割、连接等基本操作方法,正则表达式语法、正则表达式对象、子模式与 match 对象,以及 Python 正则表达式模块 re 的应用。

第 5 章　函数设计与使用。讲解函数的定义与使用,关键参数、默认值参数、长度可变参数等不同参数类型,全局变量与局部变量,参数传递时的序列解包,return 语句,lambda 表达式,等等。

第 6 章　面向对象程序设计。讲解类的定义与继承、self 与 cls 参数、类成员与实例成员、私有成员与公有成员、特殊方法与运算符重载等内容。

第 7 章　文件操作。讲解文件操作基本知识与 Python 文件对象,文本文件内容读取与写入,二进制文件操作与对象序列化,文件复制、移动、重命名、文件类型检测、MD5 值计算、压缩与解压缩等文件级操作以及目录操作有关知识。

第 8 章　异常处理结构与程序调试。讲解 Python 异常类层次结构与自定义异常类,

多种不同形式的异常处理结构,以及如何使用 IDLE 和 pdb 模块调试 Python 程序。

第 9 章 GUI 编程。讲解如何使用 wxPython 进行 GUI 编程,主要包括窗体、按钮、文本框、单选钮、复选框、组合框、列表框、树形等控件以及各种对话框的运用。

本书最大特点是信息量大、知识点紧凑、案例丰富。全书没有多余的文字和软件安装截图,充分利用宝贵的篇幅来介绍和讲解尽可能多的知识点,可以说是物超所值。本书作者具有 15 年程序设计教学经验,讲授过汇编语言、C/C++/C#、Java、PHP、Python 等多门程序设计语言,并编写过大量的应用程序。在本书内容的组织和安排上,结合了多年教学与开发过程中积累的许多案例,并巧妙地糅合进了相应的章节。

本书对 Python 内部工作原理进行了一定程度的剖析,对 Python 2.x 和 Python 3.x 之间的区别进行了深入对比和分析,并适当介绍了 Python 程序优化和安全编程的有关知识,可以满足不同层次读者的需要。

## 本书适用读者

本书可以作为(但不限于):

- 数字媒体技术、软件工程、网络工程、信息安全、会计、经济、金融、心理学、统计以及其他非计算机专业本科或专科的程序设计教材。如果作为本科非计算机专业程序设计语言公共课或选修课教材,建议采用 64 学时或 48 学时边讲边练的教学模式。
- 具有一定 Python 基础的读者进阶学习资料。
- 打算利用业余时间学习一门快乐的程序设计语言并编写几个小程序来娱乐的读者首选学习资料。
- 少数对编程具有浓厚兴趣和天赋的中学生课外阅读资料。

## 教学资源

本书提供全套教学课件、源代码、课后习题答案与分析以及授课计划和学时分配表,配套资源可以登录清华大学出版社官方网站下载或与作者联系索取,作者 QQ 号码是 306467355,微信号是 Python_dfg,电子邮箱地址是 dongfuguo2005@126.com。

由于时间仓促，作者水平有限，书中难免出现错误，不足之处还请指正并通过作者联系方式进行反馈，作者将不定期在 QQ 空间和微信发布和更新勘误表。

## 感谢

首先感谢父母的养育之恩，在当年那么艰苦的条件下还坚决支持我读书，而没有让我像其他同龄的孩子一样辍学。感谢姐姐、姐夫多年来对我的爱护以及在老家对父母的照顾，感谢善良的弟弟、弟媳在老家对父母的照顾，正是有了你们，我才能在远离家乡的城市安心工作。感谢我的妻子在生活中对我的大力支持，也感谢懂事的小女儿在我工作的时候能够在旁边安静地读书而尽量不打扰我，并在定稿前和妈妈一起帮我阅读全书并检查出了几个错别字。

感谢每一位读者，感谢您在茫茫书海中选择了本书，并衷心祝愿您能够从本书中受益，学到您需要的知识！同时也期待每一位读者的热心反馈，随时欢迎您指出书中的不足！

本书的出版获 2014 年山东省普通高校应用型人才培养专业发展支持计划项目资助。我校专业共建合作伙伴——浪潮优派科技教育有限公司总裁邵长臣先生——审阅了全书，并提出了很多宝贵的意见，在此致以诚挚的谢意。本书在编写出版过程中也得到清华大学出版社的大力支持和帮助，在此表示衷心的感谢。

董付国

于山东烟台

2015 年 5 月

# 目 录

# 第1章 基础知识

Python是一门跨平台、开源、免费的解释型高级动态编程语言，同时也支持伪编译，即将Python源程序转换为字节码来优化程序和提高运行速度，并且支持使用py2exe工具将Python程序转换为扩展名为exe的可执行程序，可以在没有安装Python解释器和相关依赖包的Windows平台上运行；Python支持命令式编程、函数式编程，完全支持面向对象程序设计，语法简洁清晰，并且拥有大量的几乎支持所有领域应用开发的成熟扩展库；Python就像胶水一样，可以把多种不同语言编写的程序融合到一起实现无缝拼接，更好地发挥不同语言和工具的优势，满足不同应用领域的需求。

## 1.1 如何选择Python版本

众所周知，Python官方网站目前同时发行Python 2.x和Python 3.x两个不同系列的版本，并且互相之间不兼容，除了输入输出方式有所不同，很多内置函数的实现和使用方式也有较大的区别，Python 3.x对Python 2.x的标准库也进行了一定程度的重新拆分和整合。在本书开始编写的时候，最新版本分别为Python 2.7.8和Python 3.4.2，本书编写完成时最新版本分别为Python 2.7.10和Python 3.4.3，并且已发布Python 3.5.0的第三个测试版。对于很多初级用户而言，最纠结的一个问题很可能是自己到底应该选择哪个版本，是选择Python 2.x还是Python 3.x，是选择Python 2.7.x还是Python 2.6.x呢？对于Python的版本演化历史，这里不多解释，需要说明的是，并不是数字越大表示版本越新，例如Python 2.7.9就比Python 3.2.6晚几个月发行，并且Python 3.2.6比Python 3.4.1也晚几个月，类似的情况还有很多。另外，虽然同系列版本中高版本比低版本更加完善和成熟，但这并不意味着最新的才是最合适的。很多扩展库的发行总是滞后于Python发行的版本，甚至目前还有很多扩展库不支持Python 3.x。因此，在选择Python的时候，一定要先考虑清楚自己学习Python的目的是什么，打算做哪方面的开发，有哪些扩展库可用，这些扩展库最高支持哪个版本的Python。这些问题全部确定以后，再做出自己的选择，这样才能事半功倍，而不至于把太多时间浪费在Python以及各种扩展库的反复安装和卸载上。同时还应该注意，当较新的Python版本推出之后，不要急于更新和替换已安装版本，而是应该在确定自己必须使用的扩展库也推出了较新版本之后再一起进行更新。

尽管如此，以目前来看Python 3.x毕竟是大势所趋，如果你暂时还没想到要做什么行业领域的应用开发，或者仅仅是为了尝试一种新的、好玩的语言，那么请毫不犹豫地选择Python 3.x系列的最高版本（目前正式发行版最高版本是Python 3.4.3）。我们相信，越来越多的扩展库将会在短时间内推出支持Python 3.x的版本。

安装好Python以后，在“开始”菜单中选择IDLE（Python GUI）命令，即可启动

Python 解释器并可以看到当前安装的 Python 版本号，如图 1-1 和图 1-2 所示。当然，如果你喜欢，也可以启动 Python(command line)来开始美妙的 Python 之旅。在 IDLE(Python GUI)和 Python(command line)两种界面中，都以三个大于号>>> 作为提示符，可以在提示符后面输入要执行的语句。在本书所有章节给出的示例代码中，>>> 符号都不需要输入，仅表示该代码是在交互模式下运行，而不带有该提示符的代码则表示是以脚本程序的方式运行的。本书主要使用 IDLE(Python GUI)环境来介绍 Python 程序的开发与应用。

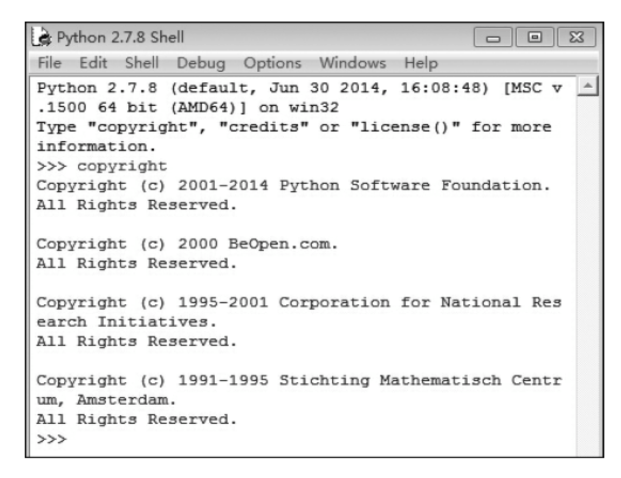



图 1-1　Python 2.7.8 主界面

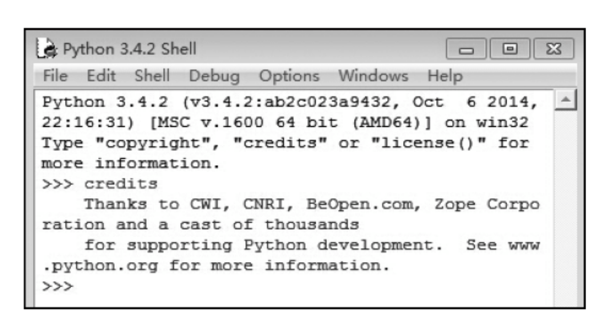



图 1-2　Python 3.4.2 主界面

除了在启动主界面上查看已安装的 Python 版本之外，还可以使用下面的命令随时进行查看。

```
>>> import sys
>>> sys.version
'3.4.2 (v3.4.2:ab2c023a9432, Oct  6 2014, 22:16:31) [MSC v.1600 64 bit (AMD64)]'
>>> sys.winver
'2.7'
>>> sys.version_info
sys.version_info(major=2, minor=7, micro=8, releaselevel='final', serial=0)
```

有时候可能需要同时安装多个不同的版本，例如，同时安装 Python 2.7.8 和 Python 3.4.2，并根据不同的开发需求在两个版本之间进行切换。多版本并存一般不影响在 IDLE 环境中直接运行程序，只需要启动相应版本的 IDLE 即可。在命令提示符环境中运行 Python 程序时，如果无法正确运行，可以尝试在调用 Python 主程序时指定其完整路径，或者通过修改系统 Path 变量来实现不同版本之间的切换。在 Windows 7 系统下修改系统 Path 变量的步骤如下：单击“开始”菜单，右击“计算机”并执行“属性”命令，在弹出的对话框中单击“高级系统设置”选项，切换至“高级”选项卡，单击“环境变量”按钮，然后修改系统 Path 变量中的 Python 安装路径，如图 1-3 所示。




图 1-3　Windows 7 环境中系统 Path 变量修改方法

## 1.2　Python 安装与简单使用

为节约篇幅，这里不再详述 Python 的安装步骤，与大多数软件的安装并没什么明显的不同，打开 Python 官方主页 https://www.python.org/后选择适合自己的版本下载并安装即可。如果使用的是 Linux 系统，例如 Ubuntu，那么很可能已经预装了某个版本的 Python，请根据需要进行升级。若未经特别说明，本书所有示例均在 Windows 7 平台上使用 Python 3.4.2 和 Python 2.7.8 进行开发和演示。

安装好以后，默认以 IDLE 为开发环境，当然也可以安装使用其他的开发环境，例如 PythonWin。本书均以 IDLE 为例，如果使用交互式编程模式，那么直接在 IDLE 提示符

>>>后面输入相应的命令并回车执行即可，如果执行顺利，马上就可以看到执行结果，否则会抛出异常。

```
>>> 3+5
8
>>> import math
>>> math.sqrt(9)
3.0
>>> 3 * (2+6)
24
>>> 2/0
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    2/0
ZeroDivisionError: integer division or modulo by zero
```

一般来讲，你可能更需要编写 Python 程序来实现特定的业务逻辑，同时也方便代码的不断完善和重复利用，毕竟直接使用交互编程模式不是很方便。在 IDLE 界面中使用菜单 File→New File 命令，创建一个程序文件，输入代码并保存为文件（务必要保证扩展名为 py，如果是 GUI 程序，可以保存为.pyw 文件。如果保存为其他扩展名的文件，一般并不影响在 IDLE 中直接运行，但是在“命令提示符”环境中运行时需要显式调用 Python 主程序，并且在资源管理器中直接双击该文件时可能会无法关联 Python 主程序，从而导致无法运行），可以使用菜单 Run→Check Module 命令来检查程序中是否存在语法错误，或者使用菜单 Run→Run Module 命令运行程序，程序运行结果将直接显示在 IDLE 交互界面上。除此之外，也可以通过在资源管理器中双击扩展名为 py 或 pyc 的 Python 程序文件直接运行；在有些情况下，可能还需要在命令提示符环境中运行 Python 程序文件。选择“开始”→“附件”→“命令提示符”命令，然后执行 Python 程序。例如，假设有程序 HelloWorld.py 内容如下。

```
def main():
    print('Hello world')
main()
```

在 IDLE 环境中运行该程序结果如图 1-4 所示。

在命令提示符环境中运行该程序的方法与结果如图 1-5 所示，该图中演示了两种执行 Python 程序的方法，虽然第二种方法看上去更简单，但是请尽量使用第一种方法来运行 Python 程序，否则可能会影响某些程序的正确运行。

```
>>> ================================ RESTART ========
=
>>>
Hello world
>>>
```

图 1-4　在 IDLE 中运行程序

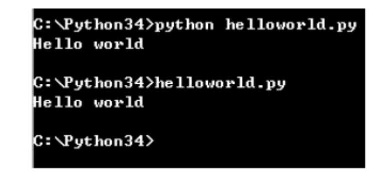



图 1-5　在命令提示符中运行程序

在实际开发中，如果用户能够熟练使用集成开发环境 IDLE 提供的一些快捷键，将会大幅度提高编写速度和开发效率。在 IDLE 环境下，除了撤销（Ctrl＋Z）、全选（Ctrl＋A）、复制（Ctrl＋C）、粘贴（Ctrl＋V）、剪切（Ctrl＋X）等常规快捷键之外，其他比较常用的快捷键如表 1-1 所示。

**表 1-1 IDLE 常用快捷键**

| 快 捷 键 | 功 能 说 明 |
| --- | --- |
| Alt＋P | 浏览历史命令（上一条） |
| Alt＋N | 浏览历史命令（下一条） |
| Ctrl＋F6 | 重启 Shell，之前定义的对象和导入的模块全部失效 |
| F1 | 打开 Python 帮助文档 |
| Alt＋/ | 自动补全前面曾经出现过的单词，如果之前有多个单词具有相同前缀，则在多个单词中循环选择 |
| Ctrl＋] | 缩进代码块 |
| Ctrl＋[ | 取消代码块缩进 |
| Alt＋3 | 注释代码块 |
| Alt＋4 | 取消代码块注释 |

## 1.3 使用 pip 管理 Python 扩展库

当前，pip 已经成为管理 Python 扩展库（或模块，一般不做区分）的主流方式，使用 pip 不仅可以实时查看本机已安装的 Python 扩展库列表，还支持纯 Python 扩展库的安装、升级和卸载等操作。使用 pip 工具管理 Python 扩展库只需要在保证计算机联网的情况下输入几个命令即可完成，极大地方便了用户。

对于 Python 2.7.9 和 Python 3.4.0 之前的版本，需要首先安装 pip 命令才能使用，而在 Python 2.7.9 以及 Python 3.4.0 之后的安装包中已经集成了该命令，不需要再单独进行安装。在较早的 Python 版本中要安装 pip，首先从 https://pypi.python.org/pypi/pip 下载文件 get-pip.py，然后在命令提示符环境中执行下面的命令：

```
python get-pip.py
```

即可自动完成 pip 的安装。当然，应保证计算机处于联网状态。

安装完成以后，就可以在命令提示符环境下使用 pip 来完成扩展库的安装、升级、卸载等操作了。如果某个模块无法使用 pip 进行安装，很可能是该模块依赖于某些动态链接库文件，此时需要登录该模块官方网站下载并单独进行安装。常用 pip 命令的使用方法如表 1-2 所示。

表 1-2 常用 pip 命令使用方法

| pip 命令示例 | 说　明 |
|---|---|
| pip install SomePackage | 安装 SomePackage 模块 |
| pip list | 列出当前已安装的所有模块 |
| pip install --upgrade SomePackage | 升级 SomePackage 模块 |
| pip uninstall SomePackage | 卸载 SomePackage 模块 |

## 1.4 Python 基础知识

本节主要介绍 Python 语言基础知识，包括对象模型、变量、运算符与表达式、内置函数以及数字、字符串等基本数据类型等。

### 1.4.1 Python 对象模型

对象是 Python 语言中最基本的概念之一。Python 中的一切都是对象，这一点可能与某些面向对象程序设计语言略有不同。Python 中有许多内置对象可供编程者直接使用，例如数字、字符串、列表、元组、字典、集合、del 命令以及 cmp()、len()、id()、type()等大量内置函数，表 1-3 中列出了其中一部分常见的 Python 对象类型；另外，有些对象需要导入特定模块(有些模块需要单独进行安装)后才能使用，如 math 模块中的正弦函数 sin()与常量 pi，random 模块中的随机数生成函数 random()，time 模块中用于返回当前时间的函数 time()，等等。

表 1-3 Python 内置对象

| 对象类型 | 示　例 | 对象类型 | 示　例 |
|---|---|---|---|
| 数字 | 1234，3.14，3＋4j | 文件 | f＝open('data.dat'，'r') |
| 字符串 | 'swfu'，"I'm student"，'''Python''' | 集合 | set('abc')，{'a'，'b'，'c'} |
| 列表 | [1，2，3]，['a'，'b'，['c'，2]] | 布尔型 | True，False |
| 字典 | {1:'food'，2:'taste'，3:'import'} | 空类型 | None |
| 元组 | (2，－5，6) | 编程单元类型 | 函数(使用 def 定义)<br>类(使用 class 定义) |

### 1.4.2 Python 变量

在 Python 中，不需要事先声明变量名及其类型，直接赋值即可创建各种类型的对象变量。例如语句

```
>>> x=3
```

创建了整型变量 x，并赋值为 3，再例如语句

```
>>> x='Hello world.'
```

创建了字符串变量 x，并赋值为'Hello world.'。这一点适用于 Python 任意类型的对象。

虽然不需要在使用之前显式地声明变量及其类型，但是 Python 仍属于强类型编程语言，Python 解释器会根据赋值或运算来自动推断变量类型。每种类型支持的运算也不完全一样，因此在使用变量时需要程序员自己确定所进行的运算是否合适，以免出现异常或者意料之外的结果。同一个运算符对于不同类型数据操作的含义和计算结果也是不一样的，后面会进行介绍。另外，Python 还是一种动态类型语言，也就是说，变量的类型是可以随时变化的，下面的代码演示了 Python 变量类型的变化。

```
>>> x=3
>>> print(type(x))
<class 'int'>
>>> x='Hello world.'
>>> print(type(x))
<class 'str'>
>>> x=[1,2,3]
>>> print(type(x))
<class 'list'>
>>> isinstance(3, int)
True
>>> isinstance('Hello world', str)
True
```

其中，内置函数 type()用来返回变量类型，内置函数 isinstance()用来测试对象是否为指定类型的实例。代码中首先创建了整型变量 x，然后又分别创建了字符串和列表类型的变量 x。当创建了字符串类型的变量 x 之后，之前创建的整型变量 x 自动失效，创建列表对象 x 之后，之前创建的字符串变量 x 自动失效。可以将该模型理解为“状态机”，在显式修改其类型或删除之前，变量将一直保持上次的类型。

在大多数情况下，如果变量出现在赋值运算符或复合赋值运算符(例如+=、*=等等)的左边则表示创建变量或修改变量的值，否则表示引用该变量的值，这一点同样适用于使用下标来访问列表、字典等可变序列以及其他自定义对象中元素的情况。例如下面的代码：

```
>>> x=3                    #创建整型变量
>>> print(x**2)
9
>>> x+=6                   #修改变量值
>>> print(x)               #读取变量值并输出显示
9
>>> x=[1,2,3]              #创建列表对象
>>> print(x)
```

```
[1, 2, 3]
>>> x[1]=5                #修改列表元素值
>>> print(x)              #输出显示整个列表
[1, 5, 3]
>>> print(x[2])           #输出显示列表指定元素
3
```

后面会提到，字符串和元组属于不可变序列，这意味着不能通过下标的方式来修改其中的元素值，例如下面的代码试图修改元组中元素的值时抛出异常。

```
>>> x=(1,2,3)
>>> print(x)
(1, 2, 3)
>>> x[1]=5
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in<module>
    x[1]=5
TypeError: 'tuple' object does not support item assignment
```

在 Python 中，允许多个变量指向同一个值，例如：

```
>>> x=3
>>> id(x)
1786684560
>>> y=x
>>> id(y)
1786684560
```

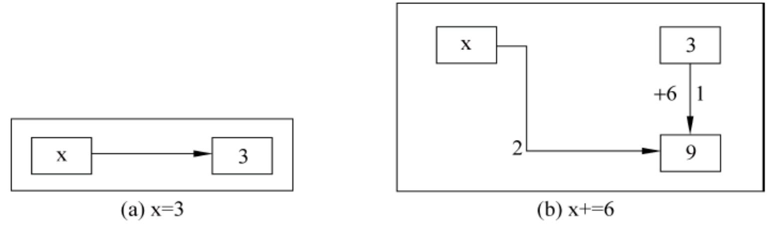
继续上面的示例代码，需要注意的是，当为其中一个变量修改值以后，其内存地址将会变化，但这并不影响另一个变量，例如，接着上面的代码再继续执行下面的代码：

```
>>> x+=6
>>> id(x)
1786684752
>>> y
3
>>> id(y)
1786684560
```

在这段代码中，内置函数 id()用来返回变量所指值的内存地址。可以看出，在 Python 中修改变量值的操作，并不是修改了变量的值，而是修改了变量指向的内存地址。这是因为 Python 解释器首先读取变量 x 原来的值，然后将其加 6，并将结果存放于内存中，最后将变量 x 指向该结果的内存空间，如图 1-6 所示。

Python 采用的是基于值的内存管理方式，如果为不同变量赋值为相同值，这个值在内存中只有一份，多个变量指向同一块内存地址，前面的几段代码也说明了这个特点。再例如下面的代码：

```
>>> x=3
>>> id(x)
10417624
>>> y=3
>>> id(y)
10417624
>>> y=5
>>> id(y)
10417600
>>> id(x)
10417624
```




(a) x=3　　(b) x+=6

图 1-6　Python 内存管理模式

Python 具有自动内存管理功能，对于没有任何变量指向的值，Python 自动将其删除。Python 会跟踪所有的值，并自动删除不再有变量指向的值。因此，Python 程序员一般情况下不需要太多考虑内存管理的问题。尽管如此，显式使用 del 命令删除不需要的值或显式关闭不再需要访问的资源，仍是一个好的习惯，同时也是一个优秀程序员的基本素养之一。

最后，在定义变量名的时候，需要注意以下问题：

- 变量名必须以字母或下划线开头，但以下划线开头的变量在 Python 中有特殊含义，本书后面第 6 章会详细讲解；
- 变量名中不能有空格以及标点符号（括号、引号、逗号、斜线、反斜线、冒号、句号、问号等等）；
- 不能使用关键字作变量名，可以导入 keyword 模块后使用 print(keyword. kwlist) 查看所有 Python 关键字；

```
>>> import keyword
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import',
'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try',
'while', 'with', 'yield']
>>> and=3
SyntaxError: invalid syntax
```

- 不建议使用系统内置的模块名、类型名或函数名以及已导入的模块名及其成员名作变量名，这将会改变其类型和含义，可以通过 dir(__builtins__)查看所有内置模块、类型和函数；
- 变量名区分英文字母的大小写，例如 student 和 Student 是不同的变量。

### 1.4.3 数字

数字属于 Python 不可变对象，1.4.2 节的例子已经说明了这一点，即修改整型变量值的时候并不是真的修改变量的值，而是先把值存放到内存中然后修改变量使其指向了新的内存地址，浮点数、复数等数字类型以及其他类型的变量具有同样的特点。列表、字典等可变类型对象的情况稍微复杂一些，将在第 2 章中详细介绍。

在 Python 中，数字类型变量可以表示任意大的数值。

```
>>> a=99999999999999999999999999999999
>>> a * a
9999999999999999999999999999999980000000000000000000000000000001L
>>> a**3
99999999999999999999999999999999700000000000000000000000000000002999999999
999999999999999999999L
```

如果你愿意，完全可以把 IDLE 当做计算器来使用，IDLE 可以实现复杂的数学运算。

```
>>> 3 * (2+5)/3.0
7.0
>>> import math                #math 是 Python 标准库，其中包含大量用于数学计算的函数
>>> math.sqrt(3**2+4**2)       #平方根
5.0
```

Python 数值类型主要有整数、浮点数和复数。整数类型主要有：

- 十进制整数，如 0、-1、9、123。
- 十六进制整数，使用 16 个数字 0、1、2、3、4、5、6、7、8、9、a、b、c、d、e、f 来表示整数，必须以 0x 开头，如 0x10、0xfa、0xabcdef。
- 八进制整数，使用 8 个数字 0、1、2、3、4、5、6、7 来表示整数，必须以 0o 开头，如 0o35、0o11。
- 二进制整数，使用 2 个数字 0、1 来表示整数，必须以 0b 开头，如 0b101、0b100。

浮点数也称小数，例如，.3、15.0、0.37、-11.2、1.2e2、314.15e-2，都是合法的浮点数。

Python 中的复数与数学中复数的形式完全一致，都是由实部和虚部构成，并且使用 j 或 J 来表示虚部。

```
>>> a=3+4j
>>> b=5+6j
>>> c=a+b
```

```
>>> c
(8+10j)
>>> c.real                #查看复数实部
8.0
>>> c.imag                #查看复数虚部
10.0
>>> a.conjugate()         #返回共轭复数
(3-4j)
>>> a*b                   #复数乘法
(-9+38j)
>>> a/b                   #复数除法
(0.6393442622950819+0.03278688524590165j)
```

### 1.4.4 字符串

在Python中，字符串属于不可变序列，一般使用单引号、双引号或三引号进行界定，并且单引号、双引号、三单引号、三双引号还可以互相嵌套，用来表示复杂字符串。例如

```
'abc'、'123'、'中国'、"Python"、'''Tom said,"Let's go"'''
```

都是合法字符串，空字符串表示为''或""或''''''，即一对不包含任何内容的任意字符串界定符。特别地，一对三单引号或三双引号表示的字符串支持换行，支持排版格式较为复杂的字符串，也可以在程序中表示较长的注释，在第4章和第5章中将分别进行介绍。

由于字符串类型应用非常广泛，其支持的操作也较多，这里先简单介绍一下，第4章再结合正则表达式全面展开进行详细讲解。

字符串支持使用+运算符进行合并以生成新字符串。

```
>>> a='abc'+'123'
>>> a
'abc123'
```

可以对字符串进行格式化，把其他类型对象按格式要求转换为字符串，并返回结果字符串，例如下面的代码：

```
>>> a=3.6674
>>> '%7.3f' %a
'  3.667'
>>> "%d:%c"%(65,65)
'65:A'
>>> """My name is %s, and my age is %d"""%('Dong Fuguo',38)
'My name is Dong Fuguo, and my age is 38'
```

Python支持转义字符，常用的转义字符如表1-4所示。

表 1-4 转义字符

| 转义字符 | 含　　义 | 转义字符 | 含　　义 |
|---|---|---|---|
| \n | 换行符 | \" | 双引号 |
| \t | 制表符 | \\ | 一个\ |
| \r | 回车 | \ddd | 3 位八进制数对应的字符 |
| \' | 单引号 | \xhh | 2 位十六进制数对应的字符 |

需要特别说明的是，字符串界定符前面加字母 r 或 R 表示原始字符串，其中的特殊字符不进行转义，但字符串的最后一个字符不能是\符号。原始字符串主要用于正则表达式，也可以用来简化文件路径或 url 的输入，请参考第 4 章的内容。

### 1.4.5 运算符与表达式

与其他语言一样，Python 支持大多数算术运算符、关系运算符、逻辑运算符以及位运算符，并遵循与大多数语言一样的运算符优先级。除此之外，还有一些运算符是 Python 特有的，例如成员测试运算符、集合运算符、同一性测试运算符等等。另外，Python 很多运算符具有多种不同的含义，作用于不同类型操作数的含义并不相同，非常灵活。常用运算符如表 1-5 所示。

表 1-5 Python 运算符

| 运算符示例 | 功 能 说 明 |
|---|---|
| x+y | 算术加法，列表、元组、字符串合并 |
| x-y | 算术减法，集合差集 |
| x*y | 乘法，序列重复 |
| x/y | 除法(在 Python 3.x 中叫做真除法) |
| x//y | 求整商 |
| -x | 相反数 |
| x%y | 余数(对实数也可以进行余数运算)，字符串格式化 |
| x**y | 幂运算 |
| x<y;x<=y;x>y;x>=y | 大小比较(可以连用)，集合的包含关系比较 |
| x==y;x!=y | 相等(值)比较，不等(值)比较 |
| x or y | 逻辑或 (只有 x 为假才会计算 y) |
| x and y | 逻辑与(只有 x 为真才会计算 y) |
| not x | 逻辑非 |
| x in y;x not in y | 成员测试运算符 |
| x is y;x is not y | 对象实体同一性测试(地址) |
| \|、^、&、<<、>>、~ | 位运算符 |
| &、\|、^ | 集合交集、并集、对称差集 |

需要说明的是，Python 中的除法有两种：/和//分别表示除法和整除运算，并且 Python 2.x 和 Python 3.x 对/运算符的解释也略有区别。Python 2.x 将/解释为普通除法，而 Python 3.x 将其解释为真除法。例如，在 Python 3.4.2 中运算结果如下：

```
>>> 3/5
0.6
>>> 3//5
0
>>> 3.0/5
0.6
>>> 3.0//5
0.0
>>> 13//10
1
>>> -13//10
-2
```

而上面的表达式在 Python 2.7.8 中运算结果如下：

```
>>> 3/5
0
>>> 3//5
0
>>> 3.0/5
0.6
>>> 3.0//5
0.0
>>> 13//10
1
>>> -13//10
-2
```

另外一个需要说明的，也是与其他有些语言略有不同的运算符是%。在 Python 中，除去前面已经介绍过的字符串格式化用法之外，该运算符还可以对整数和浮点数计算余数。但是由于浮点数的精确度影响，计算结果可能略有误差。

```
>>> 3.1%2
1.1
>>> 6.3%2.1
2.0999999999999996
>>> 6%2
0
>>> 6.0%2
0.0
>>> 6.0%2.0
0.0
```

```
>>> 5.7%4.8
0.9000000000000004
```

如前所述,Python 中很多运算符有多重含义,在程序中运算符的具体含义取决于操作数的类型,将在第 2 章中根据内容组织的需要陆续进行展开。例如 * 运算符就是Python 运算符中比较特殊的一个,它不仅可以用于数值乘法,还可以用于列表、字符串、元组等类型,当列表、字符串或元组等类型变量与整数进行 * 运算时,表示对内容进行重复并返回重复后的新对象。

```
>>> 3 * 2                  #整数相乘
6
>>> 2.0 * 3                #浮点数与整数相乘
6.0
>>> (3+4j) * 2             #复数与整数相乘
(6+8j)
>>> (3+4j) * (3-4j)        #复数与复数相乘
(25+0j)
>>> '1' * 5                #字符串重复
'11111'
>>> "a" * 10               #字符串重复
'aaaaaaaaaa'
>>> [1,2,3] * 3            #列表重复
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> (1,2,3) * 3            #元组重复
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> 3 * 'a'                #字符串重复
'aaa'
```

在 Python 中,单个任何类型的对象或常数属于合法表达式,使用表 1-5 中运算符连接的变量和常量以及函数调用的任意组合也属于合法的表达式。

```
>>> a=[1,2,3]
>>> b=[4,5,6]
>>> c=a+b
>>> c
[1, 2, 3, 4, 5, 6]
>>>d=map(str, c)     #Python 2.7.8
>>>d
['1', '2', '3', '4', '5', '6']
>>>d=list(map(str, c))     #Python 3.4.3
>>>d
['1', '2', '3', '4', '5', '6']
>>> import math
>>> map(math.sin, c)
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672, -0.7568024953079282,
-0.9589242746631385, -0.27941549819892586]
>>> 'Hello'+' '+'world'
'Hello world'
```

```
>>> 'welcome ' * 3
'welcome welcome welcome '
>>> ('welcome,' * 3).rstrip(',')+'!'
'welcome,welcome,welcome!'
```

需要注意的是，在 Python 中逗号“,”并不是运算符，而只是一个普通分隔符，例如下面的代码：

```
>>> 'a' in 'b', 'a'
(False, 'a')
>>> 'a' in ('b', 'a')
True
>>> x=3, 5
>>> x
(3, 5)
>>> 3==3, 5
(True, 5)
>>> x=3+5, 7
>>> x
(8, 7)
```

### 1.4.6 常用内置函数

内置函数是指不需要导入任何模块即可直接使用的函数，例如，在 1.4.5 节最后的例子中用到的 map()函数即属于 Python 内置函数，因此不需要导入任何模块就可以直接使用，该函数在本书后面会有讲解，当然你也可以直接跳至第 5 章进行阅读，或者使用 help(map)来查看该函数帮助文档进行学习。

执行下面的命令可以列出所有内置函数和内置对象：

```
>>> dir(__builtins__)
```

常用的内置函数及其功能简要说明如表 1-6 所示。

**表 1-6 Python 常用内置函数**

| 函 数 | 功能简要说明 |
| --- | --- |
| abs(x) | 返回数字 x 的绝对值 |
| all(iterable) | 如果对于可迭代对象中所有元素 x 都有 bool(x)为 True，则返回 True。对于空的可迭代对象也返回 True |
| any(iterable) | 只要可迭代对象中存在元素 x 使得 bool(x)为 True，则返回 True。对于空的可迭代对象，返回 False |
| bin(x) | 把数字 x 转换为二进制串 |

续表

| 函　　数 | 功能简要说明 |
| --- | --- |
| callable(object) | 测试对象是否可调用。类和函数是可调用的，包含__call__()方法的类的对象也是可调用的 |
| chr(x) | 返回 ASCII 编码为 x 的字符 |
| cmp(x,y) | 比较大小，如果 x<y 则返回负数；如果 x==y，则返回 0；如果 x>y 则返回正数。Python 3.x 不再支持该函数 |
| dir() | 返回指定对象的成员列表 |
| eval(s[,globals[,locals]]) | 计算字符串中表达式的值并返回 |
| filter(function or None,sequence) | 返回序列中使得函数值为 True 的那些元素，如果函数为 None 则返回那些值等价于 True 的元素。如果序列为元组或字符串则返回相同类型结果，其他则返回列表 |
| float(x) | 把数字或字符串 x 转换为浮点数并返回 |
| help(obj) | 返回对象 obj 的帮助信息 |
| hex(x) | 把数字 x 转换为十六进制串 |
| id(obj) | 返回对象 obj 的标识(地址) |
| input([提示内容字符串]) | 接收键盘输入的内容，返回字符串。Python 2.x 和 Python 3.x 对该函数的解释不完全一样，详见 1.4.8 节 |
| int(x[,d]) | 返回数字的整数部分，或把 d 进制的字符串 x 转换为十进制并返回，d 默认为十进制 |
| isinstance(object,class-or-type-or-tuple) | 测试对象是否属于指定类型的实例 |
| len(obj) | 返回对象 obj 包含的元素个数，适用于列表、元组、集合、字典、字符串等类型的对象 |
| list([x])、set([x])、tuple([x])、dict([x]) | 把对象转换为列表、集合、元组或字典并返回，或生成空列表、空集合、空元组、空字典 |
| map(函数,序列) | 将单参数函数映射至序列中每个元素，返回结果列表 |
| max(x)、min(x)、sum(x) | 返回序列中的最大值、最小值或数值元素之和 |
| oct(x) | 把数字 x 转换为八进制串 |
| open(name[,mode[,buffering]]) | 以指定模式打开文件并返回文件对象 |
| ord(s) | 返回 1 个字符 s 的编码 |
| pow(x,y) | 返回 x 的 y 次方，等价于 x**y |
| range([start,] end [,step] ) | 返回一个等差数列(Python 3.x 中返回一个 range 对象)，不包括终值 |
| reduce(函数,序列) | 将接收 2 个参数的函数以累积的方式从左到右依次应用至序列中每个元素，最终返回单个值作为结果 |

续表

| 函 数 | 功能简要说明 |
| --- | --- |
| reversed(列表或元组) | 返回逆序后的迭代器对象 |
| round(x [,小数位数]) | 对 x 进行四舍五入,若不指定小数位数,则返回整数 |
| str(obj) | 把对象 obj 转换为字符串 |
| sorted(列表[,cmp[,key[,reverse]]]) | 返回排序后的列表。Python 3.x 中的 sorted()方法没有 cmp 参数 |
| type(obj) | 返回对象 obj 的类型 |
| zip(seq1 [,seq2 [...]]) | 返回[(seq1[0],seq2[0] ...),(...)]形式的列表 |

由于内置函数众多且功能强大,很难一下子全部解释清楚,本书将在后面的章节中根据内容组织的需要逐步进行展开并演示其用法。这里只通过几个例子来演示部分内置函数的使用,如果需要用到某个内置函数而还没有看到本书后面的讲解,可以通过内置函数 help()查看函数的使用帮助,提前进行学习。作为一个建议,编写程序时应优先考虑使用内置函数,因为内置函数不仅成熟、稳定,而且速度相对较快。

ord()和 chr()是一对功能相反的函数,ord()用来返回单个字符的序数或 ASCII 码,而 chr()则用来返回介于 0～255 之间的某序数对应的字符,str()则直接将其任意类型参数转换为字符串。下面的代码演示了这几个函数的用法:

```
>>> ord('a')
97
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(67)
'C'
>>> chr(ord('A')+1)
'B'
>>> str(1)
'1'
>>> str(1234)
'1234'
>>> str([1,2,3])
'[1, 2, 3]'
>>> str((1,2,3))
'(1, 2, 3)'
>>>str({1,2,3})    #Python 2.7.8
'set([1, 2, 3])'
>>>str({1,2,3})    #Python 3.4.3
'{1, 2, 3}'
```

max()、min()、sum()这三个内置函数分别用于计算列表、元组或其他可迭代对象中

所有元素最大值、最小值以及所有元素之和，sum()只支持包含数值型元素的序列或可迭代对象，max()和 min()则要求序列或可迭代对象中的元素之间可比较大小。例如下面的示例代码，首先使用列表推导式生成包含 10 个随机数的列表，然后分别计算该列表的最大值、最小值和所有元素之和。

```
>>> import random
>>> a=[random.randint(1,100) for i in range(10)]
>>> a
[72, 26, 80, 65, 34, 86, 19, 74, 52, 40]
>>> print(max(a), min(a), sum(a))
86 19 548
```

很显然，如果需要计算该列表中的所有元素的平均值，可以直接使用下面的方法：

```
>>> a=[72, 26, 80, 65, 34, 86, 19, 74, 52, 40]
>>> sum(a)*1.0/len(a)          #Python 2.7.8
54.8
>>> sum(a)/len(a)              #Python 3.4.2
54.8
```

对于初学者而言，也许 dir()和 help()这两个内置函数是最有用的。使用 dir()函数可以查看指定模块中包含的所有成员或者指定对象类型所支持的操作，而 help()函数则返回指定模块或函数的说明文档，这对于了解和学习新的模块与知识是非常重要的，能够熟练使用这两个函数也是学习能力的重要体现。

下面的代码首先导入数学模块 math，然后查看该模块的常量和函数，并查看指定函数的使用帮助：

```
>>> import math
>>> dir(math)                  #查看模块中可用对象
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> help(math.sqrt)            #查看指定方法的使用帮助
Help on built-in function sqrt in module math:
sqrt(...)
    sqrt(x)
    Return the square root of x.
>>> help(math.sin)
Help on built-in function sin in module math:
sin(...)
    sin(x)
Return the sine of x (measured in radians).
```

```
>>> dir(3+4j)                          #查看数字类型对象成员
['__abs__', '__add__', '__class__', '__coerce__', '__delattr__', '__div__',
'__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__', '__format__',
'__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__int__', '__le__', '__long__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__',
'__new__', '__nonzero__', '__pos__', '__pow__', '__radd__', '__rdiv__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rmod__', '__rmul__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
'conjugate', 'imag', 'real']
>>> dir('')                            #查看字符串类型成员
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

### 1.4.7 对象的删除

正如前面所提到的，Python具有自动内存管理功能，Python解释器会跟踪所有的值，一旦发现某个值不再有任何变量指向，将会自动删除该值。尽管如此，自动内存管理或者垃圾回收机制并不能保证及时释放内存。显式释放自己申请的资源是程序员的好习惯之一，也是程序员素养的重要体现之一。

在Python中，可以使用del命令来显式删除对象并解除与值之间的指向关系。删除对象时，如果其指向的值还有别的变量指向则不删除该值，如果删除对象后该值不再有其他变量指向，则删除该值。例如下面的代码所演示：

```
>>> x=[1,2,3,4,5,6]
>>> y=3
>>> z=y
>>> print(y)
3
>>> del y                              #删除对象
>>> print(y)
Traceback (most recent call last):
```

```
  File "<pyshell#52>", line 1, in<module>
    print(y)
NameError: name 'y' is not defined
>>> print(z)
3
>>> del z
>>> print(z)
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in<module>
    print(z)
NameError: name 'z' is not defined
>>> del x[1]                        #删除列表中指定元素
>>> print(x)
[1, 3, 4, 5, 6]
>>> del x                           #删除整个列表
>>> print(x)
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in<module>
    print(x)
NameError: name 'x' is not defined
```

正如运行结果所示，变量 y 和 z 指向同一个值，删除变量 y 以后该值仍存在且被 z 所指向。另外，del 命令可以用来删除列表或其他可变序列中的指定元素，也可以删除整个列表或其他类型序列对象。列表中部分元素删除以后，列表会自动收缩其内存空间以保证各元素连续存储，这在第 2 章会详细介绍。del 命令无法删除元组或字符串中的指定元素，而只可以删除整个元组或字符串，因为这两者均属于不可变序列。

```
>>> x= (1,2,3)
>>> del x[1]
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in<module>
    del x[1]
TypeError: 'tuple' object doesn't support item deletion
>>> del x
>>> print(x)
Traceback (most recent call last):
  File "<pyshell#64>", line 1, in<module>
    print(x)
NameError: name 'x' is not defined
```

### 1.4.8 基本输入输出

在 Python 中，使用内置函数 input()来接收用户的键盘输入，input()函数的一般用法为：

```
x=input('提示：')
```

该函数返回用户输入的对象。

尽管形式一样，Python 2.x 和 Python 3.x 对该函数的解释略有不同。在 Python 2.x 中，该函数返回结果的类型由输入值时所使用的界定符来决定，例如下面的 Python 2.7.8 代码：

```
>>> x=input("Please input:")
Please input:3                  #没有界定符，整数
>>> print type(x)
<type 'int'>
>>> x=input("Please input:")
Please input:'3'                #单引号，字符串
>>> print type(x)
<type 'str'>
>>> x=input("Please input:")
Please input:[1,2,3]            #方括号，列表
>>> print type(x)
<type 'list'>
```

在 Python 2.x 中，还有另外一个内置函数 raw_input()也可以用来接收用户输入的值。与 input()函数不同的是，raw_input()函数返回结果的类型一律为字符串，而不论用户使用什么界定符。例如：

```
>>> x=raw_input("Please input:")
Please input:[1,2,3]
>>> print type(x)
<type 'str'>
```

在 Python 3.x 中，不存在 raw_input()函数，只提供了 input()函数用来接收用户的键盘输入。在 Python 3.x 中，不论用户输入数据时使用什么界定符，input()函数的返回结果都是字符串，需要将其转换为相应的类型再处理，相当于 Python 2.x 中的 raw_input()函数。例如下面的 Python 3.4.2 代码：

```
>>> x=input('Please input:')
Please input:3
>>> print(type(x))
<class 'str'>
>>> x=input('Please input:')
Please input:'1'
>>> print(type(x))
<class 'str'>
>>> x=input('Please input:')
Please input:[1,2,3]
>>> print(type(x))
```

```
<class 'str'>
>>> x=raw_input('Please input:')
Traceback (most recent call last):
  File "<pyshell#83>", line 1, in<module>
    x=raw_input('Please input:')
NameError: name 'raw_input' is not defined
```

Python 2.x 和 Python 3.x 的输出方法也不完全一致。在 Python 2.x 中,使用 print 语句进行输出,而 Python 3.x 中使用 print()函数进行输出,上面的例子已经说明了这个区别。在本书给出的代码中,大部分是在 Python 3.4.2 环境下编写的,也有少量代码是使用 Python 2.7.8 编写的,这是因为很多扩展库暂时还不支持 Python 3.x,目前仍有大量的开发人员使用 Python 2.x。若在阅读本书后面章节内容时偶尔遇到某个代码中使用 print 语句进行输出,想必你会明白原因的,并且也知道如何根据你安装的 Python 版本进行适当的改写。

默认情况下,Python 将结果输出到 IDLE 或者标准控制台,在输出时也可以进行重定向,例如可以把结果输出到指定文件。在 Python 2.7.8 中使用下面的方法进行输出重定向:

```
>>> fp=open(r'C:\mytest.txt', 'a+')
>>> print>>fp, "Hello,world"
>>> fp.close()
```

而在 Python 3.4.2 中则需要使用下面的方法进行重定向:

```
>>> fp=open(r'D:\mytest.txt', 'a+')
>>> print('Hello,world!', file=fp)
>>> fp.close()
```

另外一个重要的不同是,对于 Python 2.x 而言,在 print 语句之后加上逗号“,”则表示输出内容之后不换行,例如:

```
>>> for i in range(10):
    print i,
0 1 2 3 4 5 6 7 8 9
```

在 Python 3.x 中,为了实现上述功能则需要使用下面的方法:

```
>>> for i in range(10,20):
    print(i, end=' ')
10 11 12 13 14 15 16 17 18 19
```

在这两个示例中,range()是内置函数,用来生成一个列表或迭代对象,相信你已经了解了该函数的基本用法,更加详细和巧妙的用法将会在后面章节中逐步展开。

### 1.4.9 模块导入与使用

Python 默认安装仅包含部分基本或核心模块,但用户可以很方便地安装大量的其他

扩展模块，pip是管理扩展模块的重要工具。同样，在Python启动时，也仅加载了很少的一部分模块，在需要时由程序员显式地加载(有些模块可能需要先安装)其他模块。这样可以减小程序运行的压力，仅加载真正需要的模块和功能，且具有很强的可扩展性。可以使用sys.modules.items()显示所有预加载模块的相关信息。

正如上面所述，对于很多模块而言，需要首先导入，然后才能使用其中的对象。Python中主要有以下几种导入模块的方法。

### 1. import 模块名 [as 别名]

使用这种方式导入以后，需要在要使用的对象之前加上前缀，即以“模块名.对象名”的方式进行访问。也可以为导入的模块设置一个别名，然后可以使用“别名.对象名”的方式来使用其中的对象。

```
>>> import math
>>> math.sin(0.5)                #求 0.5 的正弦
0.479425538604203
>>> import random
>>> x=random.random()            #获得[0,1) 内的随机小数
>>> x
0.7866224717141462
>>> y=random.random()
>>> y
0.21054341257255382
>>> n=random.randint(1,100)      #获得[1,100]区间上的随机整数
>>> n
82
>>> import numpy as np           #导入模块并设置别名
>>> a=np.array((1,2,3,4))        #通过模块的别名来访问其中的对象
>>> print a
[1 2 3 4]
```

### 2. from 模块名 import 对象名[as 别名]

使用这种方式仅导入明确指定的对象，并且可以为导入的对象起一个别名。这种导入方式可以减少查询次数，提高访问速度，同时也减少了程序员需要输入的代码量，而不需要使用模块名作为前缀。例如：

```
>>> from math import sin
>>> sin(3)
0.1411200080598672
>>> from math import sin as f
>>> f(3)
0.1411200080598672
```

比较极端的情况是一次导入模块中所有对象，如

```
from math import *
```

使用这种方式固然简单省事，但是并不推荐使用，一旦多个模块中有同名的对象，这种方式将会导致混乱。

有时候在测试自己编写的模块时，可能需要频繁地修改代码并重新导入模块，在 Python 2.x 中可以使用内置方法 reload()重新导入一个模块，而在 Python 3.x 中，需要使用 imp 模块或 importlib 模块的 reload()函数。不论使用哪种方法重新加载模块，都要求该模块已经被正确加载，即第一次导入和加载模块时不能使用 reload()方法。

在导入模块时，Python 首先在当前目录中查找需要导入的模块文件，如果没有找到，则从 sys 模块的 path 变量所指定的目录中查找，如果仍没有找到模块文件，则提示模块不存在。可以使用 sys 模块的 path 变量查看 Python 导入模块时搜索模块的路径，也可以使用 append()方法向其中添加自定义的文件夹以扩展搜索路径。在导入模块时，会优先导入相应的.pyc 文件，如果相应的.pyc 文件与.py 文件时间不相符或不存在对应的.pyc 文件，则导入.py 文件并重新将该模块文件编译为.pyc 文件。关于 Python 文件名的详细介绍请参考 1.6 节的内容。

近年来，大量用于不同领域和专业的 Python 扩展库不断涌现，本书最后的附录 B 列出了其中很小一部分。在大的程序中可能会需要导入很多模块，此时应按照这样的顺序来依次导入模块：

(1) 首先导入 Python 标准库模块，例如 os、sys、re；

(2) 然后导入第三方扩展库，例如 PIL、numpy、scipy；

(3) 最后导入自己定义和开发的本地模块。

## 1.5 Python 代码编写规范

(1) 缩进。

Python 程序是依靠代码块的缩进来体现代码之间的逻辑关系的。对于类定义、函数定义、选择结构、循环结构以及异常处理结构来说，行尾的冒号以及下一行的缩进表示一个代码块的开始，而缩进结束则表示一个代码块结束了。在编写程序时，同一个级别的代码块的缩进量必须相同。例如在下面的代码中，最后一个 else 子句中的代码与其他控制结构中的代码缩进量不同，但这并不影响执行，因为在该 else 中的相同级别代码具有相同的缩进量。可以自行测试，将最后一个 else 子句中的两行代码修改为不同的缩进量，则 IDLE 会提示不正确的缩进量而拒绝执行该程序。

```
a, b, c=3, 4, 5
if a>b:
    if a>c:
        print a
    else:
```

```
        print c
else:
    if b>c:
        print b
    else:
      print c
      print 'ok'
```

尽管上面的代码可以在 Python 2.7.8(略加修改后在 Python 3.4.2)环境中正确无误地运行,但是仍建议写为下面的风格,即具有相同缩进次数的代码具有相同的缩进量。

```
a, b, c=3, 4, 5
if a>b:
    if a>c:
        print a
    else:
        print c
else:
    if b>c:
        print b
    else:
        print c
        print 'ok'
```

在 IDLE 开发环境中,一般以 4 个空格为基本缩进单位,或者使用下面的方式来修改基本缩进量,如图 1-7 所示。

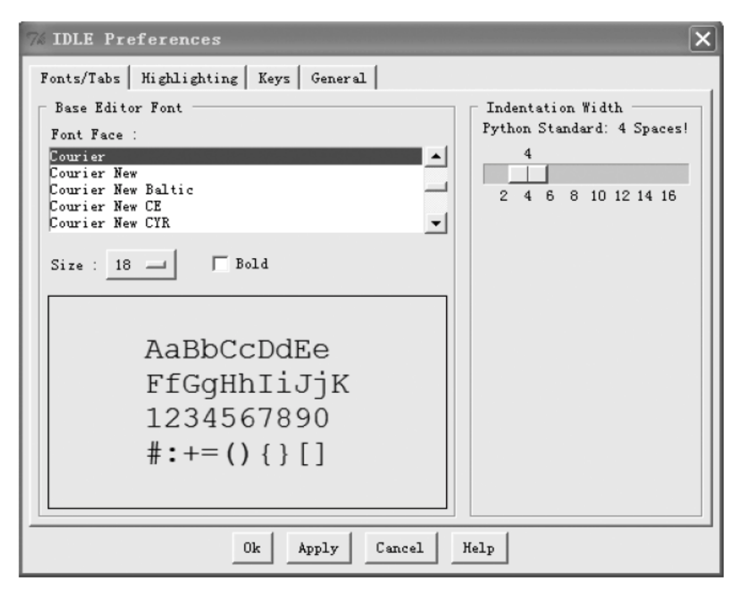



图 1-7 IDLE 环境中基本缩进量的设置

编写程序时，可以通过下面的菜单进行代码块的批量缩进和反缩进，当然需要提前使用鼠标将需要缩进或反缩进的代码块选中：

```
Fortmat==>Indent Region/Dedent Region
```

当然，也可以使用快捷键 Ctrl+]进行缩进，使用快捷键 Ctrl+[进行反缩进。

（2）注释。

注释对于程序理解和团队合作开发具有非常重要的意义。据统计，一个可维护性和可读性都很强的程序一般会包含 30%以上的注释。Python 中常用的注释方式主要有两种：

① 以符号#开始，表示本行#之后的内容为注释；

② 包含在一对三引号('''...''')或("""...""")之间且不属于任何语句的内容将被解释器认为是注释。

在 IDLE 开发环境中，可以使用鼠标选中代码块，然后使用下面的操作快速注释/解除注释代码块：

```
Format==>Comment Out Region/Uncomment Region
```

或者也使用快捷键 Alt+3 和 Alt+4 进行代码块的批量注释和解除注释。

（3）每个 import 语句只导入一个模块，尽量避免一次导入多个模块。

（4）如果一行语句太长，可以在行尾使用续行符"\"来表示下面紧接的一行仍属于当前语句，但是一般更建议使用括号来包含多行内容。

（5）使用必要的空格与空行增强代码的可读性。一般来说，运算符两侧、函数参数之间、逗号两侧建议使用空格进行分隔，而不同功能的代码块之间、不同的函数定义以及不同的类定义之间则建议增加一个空行以增加可读性。

（6）适当使用异常处理结构提高程序容错性和健壮性，但不能过多依赖异常处理结构，适当的显式判断还是必要的。

（7）软件应具有较强的可测试性，测试与开发齐头并进。

完整的 Python 编码规范请参考 PEP8(http://www.python.org/dev/peps/pep-0008)，在编写 Python 程序时，应严格遵循以上约定俗成的规范。另外，PyChecker(http://pychecker.sf.net)可以用来检查 Python 程序中的错误(bug)，并提示代码中复杂性和风格的不规范之处；Pylint(http://www.logilab.org/projects/pylint)可以用来检查模块是否符合编码规范，例如检查代码行长度、检查变量名是否符合语法规则、检查声明的接口是否全部实现，等等。

## 1.6 Python 文件名

在 Python 中，不同扩展名的文件类型有不同的含义和用途，常见的主要有以下几种：

（1）.py——Python 源文件，由 Python 解释器负责解释执行。

（2）.pyw——Python 源文件，常用于图形界面程序文件。

（3）.pyc——Python 字节码文件，无法使用文本编辑器直接查看该类型文件内容，

可用于隐藏 Python 源代码和提高运行速度。对于 Python 模块，第一次被导入时将被编译成字节码的形式，并在以后再次导入时优先使用.pyc 文件，以提高模块的加载和运行速度。对于非模块文件，直接执行时并不生成.pyc 文件，但可以使用 py_compile 模块的 compile()函数进行编译以提高加载和运行速度。另外，Python 还提供了 compileall 模块，其中包含 compile_dir()、compile_file()和 compile_path()等方法，用来支持批量 Python 源程序文件的编译。

(4) .pyo——优化的 Python 字节码文件，同样无法使用文本编辑器直接查看其内容。可以使用“python -O -m py_compile file.py”或“python -OO -m py_compile file.py”进行优化编译。

(5) .pyd——一般是由其他语言编写并编译的二进制文件，常用于实现某些软件工具的 Python 编程接口插件或 Python 动态链接库。

## 1.7 Python 脚本的__name__属性

每个 Python 脚本在运行时都有一个__name__属性。如果脚本作为模块被导入，则其__name__属性的值被自动设置为模块名；如果脚本独立运行，则其__name__属性值被自动设置为__main__。例如，假设文件 nametest.py 中只包含下面一行代码：

```
print(__name__)
```

在 IDLE 中直接运行该程序时，或者在命令行提示符环境中运行该程序文件时，运行结果如下：

```
__main__
```

而将该文件作为模块导入时得到如下执行结果：

```
>>> import nametest
nametest
```

利用__name__属性即可控制 Python 程序的运行方式。例如，编写一个包含大量可被其他程序利用的函数的模块，而不希望该模块可以直接运行，则可以在程序文件中添加以下代码：

```
if __name__=='__main__':
    print('Please use me as a module.')
```

这样一来，程序直接执行时将会得到提示“Please use me as a module.”，而使用 import 语句将其作为模块导入后可以使用其中的类、方法、常量或其他成员。

## 1.8 编写自己的包

包是 Python 用来组织命名空间和类的重要方式，可以看作是包含大量 Python 程序模块的文件夹。在包的每个目录中都必须包含一个__init__.py 文件，该文件可以是一

个空文件，仅用于表示该目录是一个包。__init__.py 文件的主要用途是设置__all__变量以及执行初始化包所需的代码，其中__all__变量中定义的对象可以在使用“from… import *”时全部被正确导入。

假设有如下结构的包：

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ⋮
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ⋮
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ⋮
```

那么，可以在自己的程序中使用下面的代码导入其中一个模块：

```
import sound.effects.echo
```

然后使用完整的名字来访问或调用其中的成员，例如：

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

或者参考 1.4.9 节中介绍的使用模块成员的方法来访问该模块中的其他成员。

## 1.9 Python 编程快速入门

在了解前面的 Python 基础知识之后，下面通过几个小程序来快速了解如何使用 Python 解决实际的问题。就像前面介绍的一样，这几个例子的代码是以脚本程序的方式给出的，所以需要在 IDLE 中创建一个程序文件，然后再输入这里的代码，最后保存为扩展名为 py 的文件并运行。

**例 1-1** 用户输入一个三位自然数，计算并输出其百位、十位和个位上的数字。

这个例子主要演示 Python 中算术运算符的用法，而计算每位上的数字有多种方法，这里只给出其中一种，你能再想出几种？哪一种方法的计算量最小？

```
x=input('请输入一个三位数：')
x=int(x)
a=x//100
b=x//10%10
c=x%10
print(a, b, c)
```

**注**：与大部分程序设计教材一样，本书中给出的代码一般都不是完整的代码，只是为了演示特定的功能用法，而没有考虑过于细节的外围工作。例如，在本例中，完整的程序还应该检查用户输入是否为数字、是否为三位数等等，可以使用“if...else...”选择结构在计算之前进行判断，也可以使用异常处理结构来增加程序的健壮性和容错性，类似问题后面不再赘述。

**例 1-2** 已知三角形的两边长及其夹角，求第三边长。

这里需要用到 math 模块中求平方根的函数 sqrt()，当然这里给出的是比较传统的写法，参考前面的知识，相信你可以写出更加简洁的代码。

```
import math
x=input('输入两边长及夹角(度)：')
a, b, theta=map(float, x.split())
c=math.sqrt(a**2+b**2-2 * a * b * math.cos(theta * math.pi/180))
print('c=', c)
```

在这段代码中使用到了序列解包的知识，在第 2 章会详细讲解，这里可以不必深究，用心体会 Python 的精妙和强大即可。

**例 1-3** 任意输入三个英文单词，按字典顺序输出。

在本例中，主要注意变量值交换的方法。

```
s=input('x,y,z=')
x, y, z=s.split(',')
if x>y:
    x, y=y, x
if x>z:
    x, z=z, x
if y>z:
    y, z=z, y
print(x, y, z)
```

**例 1-4** Python 程序框架生成器。

将下面的代码保存为 CodeFramework. py，然后在命令提示符环境中运行 CodeFramework. py newProgram. py，即可以产生 Python 程序 newProgram. py，最后使用 IDLE 打开 newProgram. py 并完善程序代码。当然，需要将下面代码中的联系方式替

换成自己的，也可以在此基础上对代码做进一步的完善和扩展。

```
import os
import sys
import datetime

head='#'+'-'*20+'\n'+\
      '#Function description:\n'+\
      '#'+'-'*20+'\n'+\
      '#Author: Dong Fuguo\n'+\
      '#QQ: 306467355\n'+\
      '#Email: dongfuguo2005@126.com\n'+\
      '#'+'-'*20+'\n'
desFile=sys.argv[1]
if os.path.exists(desFile) or not desFile.endswith('.py'):
    print('%s already exist or is not a Python code file.!'%desFile)
    sys.exit()
fp=open(desFile, 'w')
today=str(datetime.date.today().year)+'-'+str(datetime.date.today().
month)+\
       '-'+str(datetime.date.today().day)
fp.write('#-*-coding:utf-8-*-\n')
fp.write('#Filename: '+desFile+'\n')
fp.write(head)
fp.write('#Date: '+today+'\n')
fp.write('#'+'-'*20+'\n')
fp.close()
```

## 1.10 The Zen of Python

下面请大家潜心阅读“Python 之禅”，并不建议翻译下面这一段英文，这里也不打算进行过多的解读。只需要用心去体会，并在自己编写程序的时候多想想这段话，努力让自己编写的代码更加优雅、更加 Pythonic。在 IDLE 界面中可以通过执行下面的代码来获取完整的内容。

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
```

```
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one--and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea --let's do more of those!
```

## 本章小结

(1) 选择 Python 版本时应首先充分了解自己的需求和可用的扩展库情况。

(2) Python 2.x 和 Python 3.x 输入输出的方式略有不同，部分内置对象的实现和工作原理也略有不同，对标准扩展模块的组织方式也略有不同。

(3) pip 已经成为 Python 扩展库管理的标准工具。

(4) 在 Python 中一切都是对象。

(5) 在 Python 中使用变量时不需要提前声明，直接为变量赋值即可创建一个变量。

(6) Python 采用的是基于值的内存管理方式，当多个对象被赋予相同值时，该值在内存中只有一个副本。

(7) 编程时优先考虑使用内置函数来实现自己的业务逻辑。

(8) 在 Python 中，很多运算符具有多重含义。

(9) del 命令既可以删除一个变量，也可以删除列表、字典等可变序列中的部分元素。

(10) 可以使用 import 语句来导入模块中的对象，也可以为导入的模块或对象设置别名。

(11) 一般建议每个 import 语句只导入一个模块。

(12) dir()和 help()是两个非常有用的内置函数，前者可以列出指定模块或类中的对象或方法，后者可以查看相应帮助文档和使用说明。

(13) Python 程序使用缩进来体现代码之间的逻辑关系，并且建议使用必要的空格、空行和注释来提高程序的可读性。

(14) Python 程序中的注释主要有两种形式：

① 以#符号开头，表示本行该符号后的所有内容为注释；

② 放在一对三引号之间且不属于任何语句的内容被认为是注释。

(15) 可以使用异常处理结构来提高程序的健壮性，但不建议过多依赖异常处理

结构。

（16）可以通过 Python 脚本的__ name __属性来控制脚本程序的某些行为。

（17）Python 程序文件的标准扩展名为.py，Python 也支持伪编译将程序转换为字节码。

## 习题

1.1　简单说明如何选择正确的 Python 版本。

1.2　为什么说 Python 采用的是基于值的内存管理模式？

1.3　解释 Python 中的运算符/和//的区别。

1.4　在 Python 中导入模块中的对象有哪几种方式？

1.5　________是目前比较常用的 Python 扩展库管理工具。

1.6　解释 Python 脚本程序的__ name __变量及其作用。

1.7　运算符%________（可以、不可以）对浮点数进行求余数操作。

1.8　一个数字 5 ________（是、不是）合法的 Python 表达式。

1.9　在 Python 2.x 中，input()函数接收到的数据类型由________确定，而在 Python 3.x 中该函数则认为接收到的用户输入数据一律为________。

1.10　编写程序，用户输入一个三位以上的整数，输出其百位以上的数字。例如用户输入1234，则程序输出 12（提示：使用整除运算）。

# 第 2 章　Python 序列

序列是程序设计中经常用到的数据存储方式，几乎每一种程序设计语言都提供了类似的数据结构，如 C 和 Visual Basic 中的一维、多维数组等。简单地说，序列是一块用来存放多个值的连续内存空间。一般而言，在实际开发中同一个序列中的元素通常是相关的。Python 提供的序列类型可以说是所有程序设计语言类似数据结构中最灵活的，也是功能最强大的。

Python 中常用的序列结构有列表、元组、字典、字符串、集合等等。除字典和集合属于无序序列之外，列表、元组、字符串等序列类型均支持双向索引，第一个元素下标为 0，第二个元素下标为 1，以此类推；如果使用负数作为索引，则最后一个元素下标为 −1，倒数第二个元素下标为 −2，以此类推。可以使用负整数作为序列索引是 Python 语言的一大特色，熟练掌握和运用可以大幅度提高开发效率。

大量经验表明，熟练掌握 Python 基本数据结构（尤其是序列）可以更加快速有效地解决实际问题。本章通过大量案例介绍了列表、元组、字典、集合等几种基本数据结构的用法，同时还有 range()函数的巧妙应用，以及在实际应用中非常有用的列表推导式、切片操作、生成器推导式等等。在本章的最后，介绍了如何使用 Python 序列来实现栈、队列、树、图等较为复杂的数据结构并模拟其基本操作。

## 2.1　列表

列表是 Python 的内置可变序列，是包含若干元素的有序连续内存空间。在形式上，列表的所有元素放在一对方括号“[”和“]”中，相邻元素之间使用逗号分隔开。当列表增加或删除元素时，列表对象自动进行内存的扩展或收缩，从而保证元素之间没有缝隙。Python 列表内存的自动管理可以大幅度减少程序员的负担，但列表的这个特点会涉及到列表中大量元素的移动，效率较低，并且对于某些操作可能会导致意外的错误结果。因此，除非确实有必要，否则应尽量从列表尾部进行元素的增加与删除操作，这会大幅度提高列表处理速度，并且总是可以保证得到正确的结果。

在 Python 中，同一个列表中元素的数据类型可以各不相同，可以同时分别为整数、实数、字符串等基本类型，也可以是列表、元组、字典、集合以及其他自定义类型的对象。例如：

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
['spam', 2.0, 5, [10, 20]]
[['file1', 200,7], ['file2', 260,9]]
```

都是合法的列表对象。

对于 Python 序列而言，有很多方法是通用的，而不同类型的序列又有一些特有的方法。常用的列表对象方法如表 2-1 所示。除此之外，Python 的很多内置函数和命令也可以对列表和其他序列对象进行操作，后面将通过一些案例逐步进行介绍。

表 2-1 列表对象常用方法

| 方　　法 | 说　　明 |
|---|---|
| list. append(x) | 将元素 x 添加至列表尾部 |
| list. extend(L) | 将列表 L 中所有元素添加至列表尾部 |
| list. insert(index,x) | 在列表指定位置 index 处添加元素 x |
| list. remove(x) | 在列表中删除首次出现的指定元素 |
| list. pop([index]) | 删除并返回列表对象指定位置的元素，默认为最后一个元素 |
| list. clear() | 删除列表中所有元素，但保留列表对象，该方法在 Python 2. x 中没有 |
| list. index(x) | 返回第一个值为 x 的元素的下标，若不存在值为 x 的元素则抛出异常 |
| list. count(x) | 返回指定元素 x 在列表中的出现次数 |
| list. reverse() | 对列表元素进行原地翻转 |
| list. sort() | 对列表元素进行原地排序 |
| list. copy() | 返回列表对象的浅复制，该方法在 Python 2. x 中没有 |

### 2.1.1 列表创建与删除

如同其他类型的 Python 对象变量一样，使用赋值运算符“＝”直接将一个列表赋值给变量即可创建列表对象，例如：

```
>>> a_list=['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list=[]              #创建空列表
```

或者，也可以使用 list()函数将元组、range 对象、字符串或其他类型的可迭代对象类型的数据转换为列表。例如：

```
>>> a_list=list((3,5,7,9,11))
>>> a_list
[3, 5, 7, 9, 11]
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
>>> list('hello world')
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> x=list()               #创建空列表
>>> x
[]
```

上面的代码中再次用到了内置函数 range()，这是一个非常有用的函数，后面会多次用到，该函数语法为：

```
range([start,] stop[, step])
```

内置函数 range()接收 3 个参数，第一个参数表示起始值(默认为 0)，第二个参数表示终止值(结果中不包括这个值)，第三个参数表示步长(默认为 1)，该函数在 Python 3.x 中返回一个 range 可迭代对象，在 Python 2.x 中返回一个包含若干整数的列表。另外，Python 2.x 还提供了一个内置函数 xrange()(Python 3.x 中不提供该函数)，语法与 range()函数一样，但是返回 xrange 可迭代对象，类似于 Python 3.x 的 range()函数，其特点为惰性求值，而不是像 range()函数一样返回列表。例如下面的 Python 2.7.8 代码：

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xrange(10)
xrange(10)
>>> list(xrange(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

使用 Python 2.x 处理大数据或较大循环范围时，建议使用 xrange()函数来控制循环次数或处理范围，以获得更高的效率。例如，下面的 Python 2.7.8 代码对 range()和 xrange()的运行效率进行了简单的对比。

```
import time
import math

start=time.time()
for j in range(100000000):
    1+1
print time.time()-start

start=time.time()
for j in xrange(100000000):
    1+1
print time.time()-start
```

上面的代码运行结果为：

```
15.7339999676
11.7339999676
```

列表推导式也是一种常用的快速生成符合特定要求列表的方式，请参考 2.1.9 节的内容。

当不再使用时，使用 del 命令删除整个列表，如果列表对象所指向的值不再有其他对象指向，Python 将同时删除该值。

```
>>> del a_list
```

```
>>> a_list
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in<module>
    a_list
NameError: name 'a_list' is not defined
```

正如上面的代码所展示的一样，删除列表对象 a_list 之后，该对象就不存在了，再次访问时将抛出异常 NameError 提示访问的对象名不存在。

### 2.1.2 列表元素的增加

在实际应用中，列表元素的动态增加和删除也是经常遇到的操作，Python 列表提供了多种不同的方法来实现这一功能，在本节介绍如何为列表增加元素，2.1.3 节介绍删除列表元素的多种方法。

（1）可以使用＋运算符来实现将元素添加到列表中的功能。虽然这种用法在形式上比较简单也容易理解，但严格意义上来讲，这并不是真的为列表添加元素，而是创建一个新列表，并将原列表中的元素和新元素依次复制到新列表的内存空间。由于涉及大量元素的复制，该操作速度较慢，在涉及大量元素添加时不建议使用该方法。

```
>>> aList=[3,4,5]
>>> aList=aList+[7]
>>> aList
[3, 4, 5, 7]
```

（2）使用列表对象的 append()方法，原地修改列表，是真正意义上的在列表尾部添加元素，速度较快，也是推荐使用的方法。

```
>>> aList.append(9)
>>> aList
[3, 4, 5, 7, 9]
```

为了比较＋和 append()这两种方法的速度差异，请看以下代码：

```
import time

result=[]
start=time.time()
for i in range(10000):
    result=result+[i]
print(len(result), ',', time.time()-start)

result=[]
start=time.time()
for i in range(10000):
    result.append(i)
```

```
print(len(result), ',', time.time()-start)
```

在上面的代码中，分别重复执行 10 000 次＋运算和 append()方法为列表插入元素并比较这两种方法的运行时间。在代码中，使用 time 模块的 time()函数返回当前时间，然后运行代码之后计算时间差。由于各种运行时的原因，多次运行上面的代码得到的结果会有微小的差别，其中一次运行的结果如下。可以看出，这两个方法的速度相差还是非常大的，使用 append()方法比使用＋运算快约 70 倍。

```
10000, 0.21801209449768066
10000, 0.003000020980834961
```

前面曾经提到过，Python 采用的是基于值的自动内存管理方式，当为对象修改值时，并不是真的直接修改变量的值，而是使变量指向新的值，这对于 Python 所有类型的变量都是一样的，例如下面的代码：

```
>>> a=[1,2,3]
>>> id(a)
20230752
>>> a=[1,2]
>>> id(a)
20338208
```

然而，对于列表、集合、字典等可变序列类型而言，情况稍微复杂一些。以列表为例，列表中包含的是元素值的引用，而不是直接包含元素值。如果是直接修改序列变量的值，则与 Python 普通变量的情况是一样的；而如果是通过下标来修改序列中元素的值或通过可变序列对象自身提供的方法来增加和删除元素时，序列对象在内存中的起始地址是不变的，仅仅是被改变值的元素地址发生变化。例如下面的代码所示：

```
>>> a=[1,2,4]
>>> b=[1,2,3]
>>> a==b
False
>>> id(a)==id(b)
False
>>> id(a[0])==id(b[0])
True
>>> a=[1,2,3]
>>> id(a)
25289752
>>> a.append(4)
>>> id(a)
25289752
>>> a.remove(3)
>>> a
[1, 2, 4]
```

```
>>> id(a)
25289752
>>> a[0]=5
>>> a
[5, 2, 4]
>>> id(a)
25289752
```

(3) 使用列表对象的 extend()方法可以将另一个迭代对象的所有元素添加至该列表对象尾部。通过 extend()方法来增加列表元素也不改变其内存首地址，属于原地操作。例如，继续上面的代码执行下面的代码，其中 id()函数的返回结果可能与用户的执行结果不相同，这是正常的。

```
>>> a.extend([7,8,9])
>>> a
[5, 2, 4, 7, 8, 9]
>>> id(a)
25289752
>>> aList.extend([11,13])
>>> aList
[3, 4, 5, 7, 9, 11, 13]
>>> aList.extend((15,17))
>>> aList
[3, 4, 5, 7, 9, 11, 13, 15, 17]
>>> id(a)
25289752
```

(4) 使用列表对象的 insert()方法将元素添加至列表的指定位置。

```
>>> aList.insert(3,6)
>>> aList
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
```

正如本节开始所说，应尽量从列表尾部进行元素的增加与删除操作。列表的 insert()方法可以在列表的任意位置插入元素，但由于列表的自动内存管理功能，insert()方法会涉及到插入位置之后所有元素的移动，这会影响处理速度，类似的还有后面介绍的 remove()方法以及使用 pop()函数弹出列表非尾部元素和使用 del 命令删除列表非尾部元素的情况。因此，除非必要，应尽量避免在列表中间位置插入和删除元素的操作，而是优先考虑使用前面介绍的 append()方法和 2.1.3 节介绍的 pop()方法。下面的代码演示了 insert()方法和 append()方法处理速度的差别。

```
import time

def Insert():
    a=[]
```

```
    for i in range(10000):
        a.insert(0, i)

def Append():
    a=[]
    for i in range(10000):
        a.append(i)

start=time.time()
for i in range(10):
    Insert()
print 'Insert:', time.time()-start

start=time.time()
for i in range(10):
    Append()
print 'Append:', time.time()-start
```

运行结果如下，可以看到这两个方法的速度有很大差异，后面的列表元素删除方法也具有同样的特点，不再赘述。

```
Insert: 0.578000068665
Append: 0.0309998989105
```

(5) 使用乘法来扩展列表对象，将列表与整数相乘，生成一个新列表，新列表是原列表中元素的重复。

```
>>> aList=[3,5,7]
>>> bList=aList
>>> id(aList)
57091464
>>> id(bList)
57091464
>>> aList=aList * 3
>>> aList
[3, 5, 7, 3, 5, 7, 3, 5, 7]
>>> bList
[3,5,7]
>>> id(aList)
57092680
>>> id(bList)
57091464
```

从上面代码的运行结果可以看出，该操作实际上是创建了一个新列表，而不是真的扩展了原列表。该操作同样适用于字符串和元组，并具有相同的特点。

需要注意的是，当使用 * 运算符将包含列表的列表进行重复并创建新列表时，并不创

建元素的复制，而是创建已有对象的引用。因此，当修改其中一个值时，相应的引用也会被修改，例如下面的代码：

```
>>> x=[[None] * 2] * 3
>>> x
[[None, None], [None, None], [None, None]]
>>> x[0][0]=5
>>> x
[[5, None], [5, None], [5, None]]
>>> x=[[1,2,3]] * 3
>>> x[0][0]=10
>>> x
[[10, 2, 3], [10, 2, 3], [10, 2, 3]]
```

### 2.1.3 列表元素的删除

（1）使用 del 命令删除列表中的指定位置上的元素。前面已经提到过，del 命令也可以直接删除整个列表，此处不再赘述。

```
>>> a_list=[3,5,7,9,11]
>>> del a_list[1]
>>> a_list
[3, 7, 9, 11]
```

（2）使用列表的 pop()方法删除并返回指定（默认为最后一个）位置上的元素，如果给定的索引超出了列表的范围，则抛出异常。

```
>>> a_list=list((3,5,7,9,11))
>>> a_list.pop()
11
>>> a_list
[3, 5, 7, 9]
>>> a_list.pop(1)
5
>>> a_list
[3, 7, 9]
```

（3）使用列表对象的 remove()方法删除首次出现的指定元素，如果列表中不存在要删除的元素，则抛出异常。

```
>>> a_list=[3,5,7,9,7,11]
>>> a_list.remove(7)
>>> a_list
[3, 5, 9, 7, 11]
```

有时候可能需要删除列表中指定元素的所有重复，大家会很自然地想到使用"循环＋remove()"的方法，但是具体操作时很有可能会出现意料之外的错误，代码运行没有出现错误，但结果却是错的，或者代码不稳定——对某些数据处理结果是正确的，而对某些数据处理结果却是错误的。例如，下面的代码成功地删除了列表中的重复元素，执行结果是完全正确的。

```
>>> x=[1,2,1,2,1,2,1,2,1]
>>> for i in x:
    if i==1:
        x.remove(i)
>>> x
[2, 2, 2, 2]
```

然而，上面这段代码的逻辑是错误的，尽管执行结果是正确的。例如下面的代码同样试图删除列表中所有的"1"，与上面的代码完全相同，仅仅是所处理的数据发生了一点变化，然而当循环结束后却发现并没有把所有的"1"都删除，只是删除了一部分。

```
>>> x=[1,2,1,2,1,1,1]
>>> for i in x:
    if i==1:
        x.remove(i)
>>> x
[2, 2, 1]
```

很容易看出，两组数据的本质区别在于，第一组数据中没有连续的"1"，而第二组数据中存在连续的"1"。出现这个问题的原因是列表的自动内存管理功能。前面已经提到，在删除列表元素时，Python 会自动对列表内存进行收缩并移动列表元素以保证所有元素之间没有空隙，增加列表元素时也会自动扩展内存并对元素进行移动以保证元素之间没有空隙。每当插入或删除一个元素之后，该元素位置后面所有元素的索引就都改变了。下面的代码很好地说明了这个问题：

```
>>> x=list(range(20))
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> for i in range(len(x)):
    del x[0]
>>> x
[]
```

为了更清楚地解释这个问题带来的影响，对上面最开始给出的代码进行适当插桩，以便了解具体的执行过程。

```
>>> x=[1,2,1,2,1,1,1]
>>> for i in x:
    i
```

```
    if i==1:
        x.remove(i)
        x
1
[2, 1, 2, 1, 1, 1]
1
[2, 2, 1, 1, 1]
1
[2, 2, 1, 1]
1
[2, 2, 1]
```

上面这段代码的执行过程如图 2-1 所示。

x [1, 2, 1, 2, 1, 1, 1]
x [2, 1, 2, 1, 1, 1]
x [2, 2, 1, 1, 1]
x [2, 2, 1, 1]
x [2, 2, 1]

图 2-1　代码运行过程示意图

```
>>> x=[1,2,1,2,1,1,1]
>>> for i in x[::]:
    i
    if i==1:
        x.remove(i)
        x
1
[2, 1, 2, 1, 1, 1]
2
1
[2, 2, 1, 1, 1]
2
1
[2, 2, 1, 1]
1
[2, 2, 1]
1
[2, 2]
```

上面这段代码的执行过程如图 2-2 所示。

x[::] [1, 2, 1, 2, 1, 1, 1]
x [1, 2, 1, 2, 1, 1, 1]
x[::] [1, 2, 1, 2, 1, 1, 1]
x [2, 1, 2, 1, 1, 1]
x[::] [1, 2, 1, 2, 1, 1, 1]
x [2, 1, 2, 1, 1, 1]
x[::] [1, 2, 1, 2, 1, 1, 1]
x [2, 2, 1, 1, 1]
x[::] [1, 2, 1, 2, 1, 1, 1]
x [2, 2, 1, 1, 1]
x[::] [1, 2, 1, 2, 1, 1, 1]
x [2, 2, 1, 1]
x[::] [1, 2, 1, 2, 1, 1, 1]
x [2, 2, 1]
x[::] [1, 2, 1, 2, 1, 1, 1]
x [2, 2]

图 2-2　代码执行过程示意图

```
>>> x=[1,2,1,2,1,1,1]
>>> for i in x[::-1]:
    i
    if i==1:
        x.remove(i)
        x
1
[2, 1, 2, 1, 1, 1]
1
[2, 2, 1, 1, 1]
1
```

```
[2, 2, 1, 1]
2
1
[2, 2, 1]
2
1
[2, 2]
```

关于切片的讲解可以参考 2.1.6 节。另外,也可以使用从后向前的顺序来删除列表中的重复元素,例如下面的代码所示:

```
>>> x= [1,2,1,2,1,1,1]
>>> for i in range(len(x)-1,-1,-1):
    i
    if x[i]==1:
        del x[i]
        x
6
[1, 2, 1, 2, 1, 1]
5
[1, 2, 1, 2, 1]
4
[1, 2, 1, 2]
3
2
[1, 2, 2]
1
0
[2, 2]
```

如果使用从前向后的顺序则会出错,例如下面的代码,具体原因可以参考图 2-1 的分析:

```
>>> x= [1,2,1,2,1,1,1]
>>> for i in range(len(x)):
    i
    if x[i]==1:
        del x[i]
        x
0
[2, 1, 2, 1, 1, 1]
1
[2, 2, 1, 1, 1]
2
[2, 2, 1, 1]
3
```

```
[2, 2, 1]
4
Traceback (most recent call last):
  File "<pyshell#108>", line 3, in<module>
    if x[i]==1:
IndexError: list index out of range
```

### 2.1.4 列表元素访问与计数

如同其他语言里的数组一样，可以使用下标直接访问列表中的元素。如果指定下标不存在，则抛出异常提示下标越界，例如：

```
>>> aList=[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[3]
6
>>> aList[3]=5.5
>>> aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> aList[15]
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in<module>
    aList[15]
IndexError: list index out of range
```

使用列表对象的 index()方法可以获取指定元素首次出现的下标，语法为 index(value, [start, [stop]])，其中 start 和 stop 用来指定搜索范围，start 默认为 0，stop 默认为列表长度。若列表对象中不存在指定元素，则抛出异常提示列表中不存在该值，例如：

```
>>> aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> aList.index(7)
4
>>> aList.index(100)
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in<module>
    aList.index(100)
ValueError: 100 is not in list
```

如果需要知道指定元素在列表中出现的次数，可以使用列表对象的 count()方法进行统计，例如下面的代码：

```
>>> aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> aList.count(7)
1
```

```
>>> aList.count(0)
0
```

该方法也可以用于元组、字符串以及 range 对象,例如:

```
>>> range(10).count(3)
1
>>> (3,3,3,4).count(3)
3
>>> 'abcdefgabc'.count('abc')
2
```

### 2.1.5 成员资格判断

如果需要判断列表中是否存在指定的值,可以使用前面介绍的 count()方法,如果存在,则返回大于0的数,如果返回0,则表示不存在。或者,使用更加简洁的 in 关键字来判断一个值是否存在于列表中,返回结果为 True 或 False。

```
>>> aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> 3 in aList
True
>>> 18 in aList
False
>>> bList=[[1], [2], [3]]
>>> 3 in bList
False
>>> 3 not in bList
True
>>> [3] in bList
True
>>> aList=[3, 5, 7, 9, 11]
>>> bList=['a', 'b', 'c', 'd']
>>> (3, 'a') in zip(aList, bList)
True
>>> for a, b in zip(aList, bList):
    print(a, b)
3 a
5 b
7 c
9 d
```

关键字 in 和 not in 也可以用于其他可迭代对象,包括元组、字典、range 对象、字符串、集合等等,常用在循环语句中对序列或其他可迭代对象中的元素进行遍历,在前面的

例子中已经见过这个用法，后面的章节中还会多次用到。使用这种方法来遍历序列或迭代对象，可以减少代码的输入量、简化程序员的工作，并且大幅度提高程序的可读性，建议熟练掌握和运用。

### 2.1.6 切片操作

切片是 Python 序列的重要操作之一，适用于列表、元组、字符串、range 对象等类型。切片使用 2 个冒号分隔的 3 个数字来完成：第一个数字表示切片开始位置(默认为 0)，第二个数字表示切片截止(但不包含)位置(默认为列表长度)，第三个数字表示切片的步长(默认为 1)，当步长省略时可以顺便省略最后一个冒号。可以使用切片来截取列表中的任何部分，得到一个新列表，也可以通过切片来修改和删除列表中部分元素，甚至可以通过切片操作为列表对象增加元素。

与使用下标访问列表元素的方法不同，切片操作不会因为下标越界而抛出异常，而是简单地在列表尾部截断或者返回一个空列表，代码具有更强的健壮性。

```
>>> aList=[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::]
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::-1]
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> aList[::2]
[3, 5, 7, 11, 15]
>>> aList[1::2]
[4, 6, 9, 13, 17]
>>> aList[3::]
[6, 7, 9, 11, 13, 15, 17]
>>> aList[3:6]
[6, 7, 9]
>>> aList[3:6:1]
[6, 7, 9]
>>> aList[0:100:1]
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> a[100:]
[]
```

可以使用切片操作来快速实现很多目的，例如原地修改列表内容，列表元素的增、删、改、查以及元素替换等操作都可以通过切片来实现，并且不影响列表对象内存地址。

```
>>> aList=[3, 5, 7]
>>> aList[len(aList):]
[]
>>> aList[len(aList):]=[9]
>>> aList
```

```
[3, 5, 7, 9]
>>> aList[:3]=[1, 2, 3]
>>> aList
[1, 2, 3, 9]
>>> aList[:3]=[]
>>> aList
[9]
>>> aList=list(range(10))
>>> aList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> aList[::2]=[0] * (len(aList)//2)
>>> aList
[0, 1, 0, 3, 0, 5, 0, 7, 0, 9]
```

也可以结合使用 del 命令与切片操作来删除列表中的部分元素。

```
>>> aList=[3, 5, 7, 9, 11]
>>> del aList[:3]
>>> aList
[9, 11]
```

需要注意的是,切片返回的是列表元素的浅复制,与列表对象的直接赋值并不一样。例如下面的代码:

```
>>> aList=[3, 5, 7]
>>> bList=aList             #bList 与 aList 指向同一块内存
>>> bList
[3, 5, 7]
>>> bList[1]=8
>>> aList
[3, 8, 7]
>>> aList==bList
True
>>> aList is bList
True
>>> id(aList)               #这里的输出结果很可能和你的不一样,这是正常的
19061816
>>> id(bList)
19061816
>>> aList=[3, 5, 7]
>>> bList=aList[::]         #浅复制
>>> aList==bList
True
>>> aList is bList
False
>>> id(aList)==id(bList)
```

```
False
>>> bList[1]=8
>>> bList
[3, 8, 7]
>>> aList
[3, 5, 7]
>>> aList==bList
False
>>> aList is bList
False
>>> id(aList)
19061816
>>> id(bList)
11656168
>>> cmp(aList, bList)                        #内置函数 cmp()的知识请参考 2.1.8 节
-1
```

### 2.1.7 列表排序

在实际应用中，经常需要对列表元素进行排序，一个很自然的想法就是使用列表对象自身提供的 sort()方法进行原地排序，该方法支持多种不同的排序方式。

```
>>> aList=[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> import random
>>> random.shuffle(aList)                    #打乱顺序
>>> aList
[3, 4, 15, 11, 9, 17, 13, 6, 7, 5]
>>> aList.sort()                             #默认为升序排列
>>> aList
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList.sort(reverse=True)                 #降序排列
>>> aList
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> aList.sort(key=lambda x: len(str(x)))    #自定义排序
>>> aList
[9, 7, 6, 5, 4, 3, 17, 15, 13, 11]
```

也可以使用内置函数 sorted()对列表进行排序，与列表对象的 sort()方法不同，内置函数 sorted()返回新列表，并不对原列表进行任何修改。

```
>>> aList
[9, 7, 6, 5, 4, 3, 17, 15, 13, 11]
>>> sorted(aList)
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
```

```
>>> sorted(aList, reverse=True)              #降序排列
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
```

在某些应用中可能需要将列表元素进行逆序排列，也就是所有元素位置反转，第一个元素与最后一个元素交换位置，第二个元素与倒数第二个元素交换位置，以此类推。可以使用列表对象的 reverse()方法将所有元素原地逆序：

```
>>> import random
>>> aList=[random.randint(50, 100) for i in range(10)]
>>> aList
[87, 79, 52, 96, 56, 59, 74, 80, 53, 79]
>>> aList.reverse()
>>> aList
[79, 53, 80, 74, 59, 56, 96, 52, 79, 87]
```

Python 提供了内置函数 reversed()支持对列表元素进行逆序排列，与列表对象的 reverse()方法不同，内置函数 reversed()不对原列表做任何修改，而是返回一个逆序排列后的迭代对象。例如：

```
>>> aList=[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> newList=reversed(aList)
>>> newList
<listreverseiterator object at 0x0000000003624198>
>>> list(newList)
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> for i in newList:
    print(i, end=' ')
```

上面代码中最后的 for 循环没有输出任何内容，因为在之前的 list()函数执行时，迭代对象已遍历结束，需要重新创建迭代对象才能再次访问其内容。即：

```
>>> newList=reversed(aList)
>>> for i in newList:
    print(i, end=' ')
17 15 13 11 9 7 6 5 4 3
```

### 2.1.8 用于序列操作的常用内置函数

由于序列的重要性和广泛应用，Python 提供了大量可用于序列操作的内置函数，在 1.4.6 节中已经介绍了几个，本节再通过一些示例来简单扩展一下。

(1) cmp(序列 1，序列 2)：对两个列表进行比较，若第一个列表大于第二个，则结果为 1，相反则为－1，元素完全相同则结果为 0，类似于＝＝、＞、＜等关系运算符，但和 is、is not 不一样。例如：

```
>>> (1, 2, 3)<(1, 2, 4)
```

```
True
>>> cmp((1, 2, 3), (1, 2, 4))
-1
>>> [1, 2, 3]<[1, 2, 4]
True
>>> 'ABC'<'C'<'Pascal'<'Python'
True
>>> cmp('Pascal', 'Python')
-1
>>> (1, 2, 3, 4)<(1, 2, 4)
True
>>> (1, 2)<(1, 2, -1)
True
>>> cmp((1, 2), (1, 2, -1))
-1
>>> (1, 2, 3)==(1.0, 2.0, 3.0)
True
>>> cmp((1, 2, 3), (1.0, 2.0, 3.0))
0
>>> (1, 2, ('aa', 'ab'))<(1, 2, ('abc', 'a'), 4)
True
>>> cmp('a', 'A')
1
>>> 'a'>'A'
True
```

在 Python 3.x 中，不再支持 cmp()函数，可以直接使用关系运算符来比较数值或序列的大小，也可以使用对象的__le__()及其相关方法，或者也可以使用其他写法来模拟 Python 2.x 的内置函数 cmp()，下面代码分别进行了演示。

```
>>> a=[1, 2]
>>> b=[1, 2, 3]
>>> (a>b)-(a<b)
-1
>>> a.__le__(b)
True
>>> a.__gt__(b)
False
>>> a>b
False
>>> a<b
True
```

(2) len(列表)：返回列表中的元素个数，同样适用于元组、字典、集合、字符串、range 对象等各种可迭代对象。

(3) max(列表)、min(列表):返回列表中的最大或最小元素,同样适用于元组、字符串、集合、range 对象、字典等等,要求所有元素之间可以进行大小比较。另外,对字典进行操作时,默认是对字典的“键”进行计算,如果需要对字典“值”进行计算,则需要使用字典对象的 values()方法明确说明。

```
>>> a={1:1, 2:5, 3:8}
>>> max(a)
3
>>> max(a.values())
8
```

(4) sum(列表):对数值型列表的元素进行求和运算,对非数值型列表运算则出错,同样适用于数值型元组、集合、range 对象、字典等等。对字典进行操作时,默认是对字典“键”进行计算,如果需要对字典“值”进行计算,则需要使用字典对象的 values()方法明确说明。

```
>>> a={1:1, 2:5, 3:8}
>>> sum(a)
6
>>> sum(a.values())
14
```

(5) zip(列表 1,列表 2,…):将多个列表或元组对应位置的元素组合为元组,并返回包含这些元组的列表(Python 2.x)或 zip 对象(Python 3.x)。例如在 Python 2.7.8 中代码运行如下:

```
>>> aList=[1, 2, 3]
>>> bList=[4, 5, 6]
>>> cList=[7, 8, 9]
>>> dList=zip(aList, bList, cList)
>>> dList
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

而在 Python 3.4.2 中则需要这样使用:

```
>>> aList=[1, 2, 3]
>>> bList=[4, 5, 6]
>>> cList=zip(a, b)
>>> cList
<zip object at 0x0000000003728908>
>>> list(cList)
[(1, 4), (2, 5), (3, 6)]
```

(6) enumerate(列表):枚举列表、元组或其他可迭代对象的元素,返回枚举对象,枚举对象中每个元素是包含下标和元素值的元组。该函数对字符串、字典同样有效。但对字典进行操作时,默认是对字典“键”进行计算,如果需要对字典“值”进行枚举,则需要使

用字典对象的 values()方法明确说明。

```
>>> for item in enumerate(cList):
    print(item)
(0, (1, 4))
(1, (2, 5))
(2, (3, 6))
>>> for index, ch in enumerate('SDIBT'):
    print((index, ch), end=',')
(0, 'S'),(1, 'D'),(2, 'I'),(3, 'B'),(4, 'T'),
>>> a
{1: 1, 2: 5, 3: 8}
>>> for i, v in enumerate(a):
    print(i, v)
0 1
1 2
2 3
>>> for i, v in enumerate(a.values()):
    print(i, v)
0 1
1 5
2 8
```

### 2.1.9 列表推导式

列表推导式可以说是 Python 程序开发时应用最多的技术之一。2.1.7 节曾经使用列表推导式来快速生成包含多个随机数的列表，可以看出，列表推导式使用非常简洁的方式来快速生成满足特定需求的列表，代码具有非常强的可读性。例如：

```
>>> aList=[x * x for x in range(10)]
```

相当于

```
>>> aList=[]
>>> for x in range(10):
    aList.append(x * x)
```

而

```
>>> freshfruit=[' banana', ' loganberry ', 'passion fruit ']
>>> aList=[w.strip() for w in freshfruit]
```

则等价于下面的代码：

```
>>> freshfruit=[' banana', ' loganberry ', 'passion fruit ']
>>> for i, v in enumerate(freshfruit):
```

```
    freshfruit[i]=v.strip()
```

同时,也等价于

```
>>> freshfruit=[' banana', ' loganberry ', 'passion fruit ']
>>> freshfruit=list(map(str.strip, freshfruit))
```

但是不等价于下面的代码:

```
>>> freshfruit=[' banana', ' loganberry ', 'passion fruit ']
>>> for i in freshfruit:
    i=i.strip()
```

接下来再通过几个示例来进一步展示列表推导式的强大功能。

(1) 使用列表推导式实现嵌套列表的平铺。

```
>>> vec=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(2) 过滤不符合条件的元素。

在列表推导式中可以使用 if 子句来进行筛选,只在结果列表中保留符合条件的元素。例如,下面的代码可以列出当前文件夹下所有 Python 源文件:

```
>>> import os
>>> [filename for filename in os.listdir('.') if filename.endswith('.py')]
```

而下面的代码用于从列表中选择符合条件的元素组成新的列表:

```
>>> aList=[-1, -4, 6, 7.5, -2.3, 9, -11]
>>> [i for i in aList if i>0]
[6, 7.5, 9]
```

再例如,已知有一个包含一些同学成绩的字典,计算成绩的最高分、最低分和平均分,并查找所有最高分同学,代码可以编写如下:

```
>>> scores={"Zhang San": 45, "Li Si": 78, "Wang Wu": 40, "Zhou Liu": 96, "Zhao
Qi": 65, "Sun Ba": 90, "Zheng Jiu": 78, "Wu Shi": 99, "Dong Shiyi": 60}
>>> highest=max(scores.values())
>>> lowest=min(scores.values())
>>> highest
99
>>> lowest
40
>>> average=sum(scores.values())*1.0/len(scores)        #Python 2.7.8
>>> average
72.33333333333333
>>> highestPerson=[name for name, score in scores.items() if score==highest]
>>> highestPerson
```

```
['Wu Shi']
```

(3) 在列表推导式中使用多个循环,实现多序列元素的任意组合,并且可以结合条件语句过滤特定元素。

```
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x !=y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

(4) 使用列表推导式实现矩阵转置。

```
>>> matrix=[ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

也可以使用内置函数 zip()和 list()来实现矩阵转置:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

(5) 列表推导式中可以使用函数或复杂表达式。

```
>>> def f(v):
    if v%2==0:
        v=v**2
    else:
        v=v+1
    return v


>>> print([f(v) for v in [2, 3, 4, -1] if v>0])
[4, 4, 16]
>>> print([v**2 if v%2==0 else v+1 for v in [2, 3, 4, -1] if v>0])
[4, 4, 16]
```

(6) 列表推导式支持文件对象迭代。

```
>>> fp=open('C:\install.log', 'r')
>>> print([line for line in fp]) #为节约篇幅,这里没有给出代码运行结果
>>> fp.close()
```

(7) 使用列表推导式生成 100 以内的所有素数。

```
>>> [p for p in range(2, 100) if 0 not in [p%d for d in range(2, int(sqrt(p))+1)]]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97]
```

# 2.2 元组

与列表类似，元组也是 Python 的一个重要序列结构，但与列表不同的是，元组属于不可变序列。元组一旦创建，用任何方法都不可以修改其元素的值，也无法为元组增加或删除元素，如果确实需要修改的话，只能再创建一个新的元组。

元组的定义形式和列表相似，区别在于定义元组时所有元素放在一对圆括号"("和")"中，而不是方括号中。

## 2.2.1 元组的创建与删除

使用"="将一个元组赋值给变量，就可以创建一个元组变量。

```
>>> a_tuple=('a', )
>>> a_tuple
('a')
>>> a_tuple=('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> x=() #空元组
>>> x
()
```

如果要创建只包含一个元素的元组，只把元素放在圆括号里是不行的，还需要在元素后面加一个逗号","，而创建包含多个元素的元组则没有这个限制。

```
>>> a=3
>>> a
3
>>> a=(3)
>>> a
3
>>> a=3,
>>> a
(3,)
>>> a=1, 2
>>> a
(1, 2)
```

如同使用 list()函数将序列转换为列表一样，也可以使用 tuple()函数将其他类型序列转换为元组。

```
>>> print(tuple('abcdefg'))
('a', 'b', 'c', 'd', 'e', 'f', 'g')
```

```
>>> aList
[-1, -4, 6, 7.5, -2.3, 9, -11]
>>> tuple(aList)
(-1, -4, 6, 7.5, -2.3, 9, -11)
>>> s=tuple() #空元组
>>> s
()
```

对于元组而言，只能使用 del 命令删除整个元组对象，而不能只删除元组中的部分元素，因为元组属于不可变序列。

### 2.2.2 元组与列表的区别

列表属于可变序列，可以随意地修改列表中的元素值以及增加和删除列表元素，而元组属于不可变序列，元组中的数据一旦定义就不允许通过任何方式进行更改。因此，元组没有提供 append()、extend()和 insert()等方法，无法向元组中添加元素；同样，元组也没有 remove()和 pop()方法，也不支持对元组元素进行 del 操作，不能从元组中删除元素，只能使用 del 命令删除整个元组。元组也支持切片操作，但是只能通过切片来访问元组中的元素，而不支持使用切片来修改元组中元素的值，也不支持使用切片操作来为元组增加或删除元素。

Python 内置函数 tuple()可以接收一个列表、字符串或其他序列类型和迭代器作为参数，并返回一个包含同样元素的元组，而 list()函数可以接收一个元组、字符串或其他序列类型和迭代器作为参数并返回一个列表。从效果上看，tuple()函数可以看作是在冻结列表并使其不可变，而 list()函数是在融化元组使其可变。

元组的访问和处理速度比列表更快。如果定义了一系列常量值，主要用途仅是对它们进行遍历或其他类似用途，而不需要对其元素进行任何修改，那么一般建议使用元组而不用列表。可以认为元组对不需要修改的数据进行了“写保护”，从内在实现上不允许修改其元素值，从而使得代码更加安全。

另外，作为不可变序列，与整数、字符串一样，元组可用作字典的键，而列表则永远都不能当做字典键使用，因为列表不是不可变的。

最后，虽然元组属于不可变序列，其元素的值是不可改变的，但是如果元组中包含可变序列，情况略有不同，例如下面的代码：

```
>>> x=([1, 2], 3)
>>> x[0][0]=5
>>> x
([5, 2], 3)
>>> x[0].append(8)
>>> x
([5, 2, 8], 3)
>>> x[0]=x[0]+[10]
```

```
Traceback (most recent call last):
  File "<pyshell#83>", line 1, in<module>
    x[0]=x[0]+[10]
TypeError: 'tuple' object does not support item assignment
>>> x
([5, 2, 8], 3)
```

### 2.2.3 序列解包

在实际开发中，序列解包是非常重要和常用的一个用法，可以使用非常简洁的形式完成复杂的功能，大幅度提高了代码的可读性，并且减少了程序员的代码输入量。例如，可以使用序列解包功能对多个变量同时进行赋值：

```
>>> x, y, z=1, 2, 3
>>> x, y, z
(1, 2, 3)
>>> print(x, y, z)
1 2 3
```

再例如

```
>>> v_tuple=(False, 3.5, 'exp')
>>> (x, y, z)=v_tuple
```

或者

```
>>> x, y, z=v_tuple
```

序列解包也可以用于列表和字典，但是对字典使用时，默认是对字典“键”进行操作，如果需要对“键-值对”进行操作，需要使用字典的 items()方法说明，如果需要对字典“值”进行操作，则需要使用字典的 values()方法明确指定。对字典操作时，不需要对元素的顺序考虑过多。下面的代码演示了列表与字典的序列解包操作：

```
>>> a=[1, 2, 3]
>>> b, c, d=a
>>> s={'a':1, 'b':2, 'c':3}
>>> b, c, d=s.items()
>>> b
('c', 3)
>>> b, c, d=s
>>> b
'c'
>>> b, c, d=s.values()
>>> print(b, c, d)
1 3 2
```

使用序列解包可以很方便地同时遍历多个序列。

```
>>> keys=['a', 'b', 'c', 'd']
>>> values=[1, 2, 3, 4]
>>> for k, v in zip(keys, values):
    print(k, v)
a 1
b 2
c 3
d 4
```

在 2.1.8 节中关于内置函数 enumerate()的示例中,也是采用了序列解包的操作。再例如,下面对字典的操作也使用到了序列解包:

```
>>> s={'a':1, 'b':2, 'c':3}
>>> for k, v in s.items():
    print(k, v)

a 1
c 3
b 2
```

在调用函数时,在实参前面加上一个星号“ * ”也可以进行序列解包,从而实现将序列中的元素值依次传递给相同数量的形参,详见 5.3.4 节的讨论。

### 2.2.4 生成器推导式

从形式上看,生成器推导式与列表推导式非常接近,只是生成器推导式使用圆括号而不是列表推导式所使用的方括号。与列表推导式不同的是,生成器推导式的结果是一个生成器对象,而不是列表,也不是元组。使用生成器对象的元素时,可以根据需要将其转化为列表或元组,也可以使用生成器对象的 next()方法(Python 2.x)或__next__()方法(Python 3.x)进行遍历,或者直接将其作为迭代器对象来使用。但是不管用哪种方法访问其元素,当所有元素访问结束以后,如果需要重新访问其中的元素,必须重新创建该生成器对象。

```
>>> g=((i+2)**2 for i in range(10))
>>> g
<generator object<genexpr>at 0x02B15C60>
>>> tuple(g)                          #转化为元组
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
>>> tuple(g)                          #元素已经遍历结束
()
>>> g=((i+2)**2 for i in range(10)) #重新创建生成器对象
>>> list(g)                           #转化为列表
```

```
[4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>> g= ((i+2)**2 for i in range(10))
>>> g.next()                          #单步迭代,在 Python 3.x 中应改为__next__()
4
>>> g.next()
9
>>> g.next()
16
>>> g.next()
25
>>> g= ((i+2)**2 for i in range(10))
>>> for i in g:                       #直接进行循环迭代
    print i,
4 9 16 25 36 49 64 81 100 121
```

## 2.3 字典

字典是“键-值对”的无序可变序列,字典中的每个元素包含两部分:“键”和“值”。定义字典时,每个元素的“键”和“值”用冒号分隔,相邻元素之间用逗号分隔,所有的元素放在一对大括号“{”和“}”中。

字典中的“键”可以是 Python 中任意不可变数据,例如整数、实数、复数、字符串、元组等等,但不能使用列表、集合、字典作为字典的“键”,因为这些类型的对象是可变的。另外,字典中的“键”不允许重复,而“值”是可以重复的。

可以使用内置函数 globals()返回和查看包含当前作用域内所有全局变量和值的字典,使用内置函数 locals()返回包含当前作用域内所有局部变量和值的字典。

```
>>> a= (1, 2, 3, 4, 5)                #全局变量
>>> b= 'Hello world.'                 #全局变量
>>> def demo():
    a=3                               #局部变量
    b= [1, 2, 3]                      #局部变量
    print('locals:', locals())
    print('globals:', globals())
>>> demo()
locals: {'a': 3, 'b': [1, 2, 3]}
globals: {'a': (1, 2, 3, 4, 5), 'b': 'Hello world.', '__builtins__':<module
'__builtin__' (built-in)>, 'demo':<function demo at 0x013907F0>, '__package__':
None, '__name__': '__main__', '__doc__': None}
```

### 2.3.1 字典创建与删除

与列表和元组的创建一样,使用“=”将一个字典赋值给一个变量即可创建一个字典

变量。

```
>>> a_dict={'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict
{'database': 'mysql', 'server': 'db.diveintopython3.org'}
```

可以使用内置函数 dict()通过已有数据快速创建字典：

```
>>> keys=['a', 'b', 'c', 'd']
>>> values=[1, 2, 3, 4]
>>> dictionary=dict(zip(keys, values))
>>> print(dictionary)
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
>>> x=dict()                          #空字典
>>> x
{}
>>> type(x)
<type 'dict'>
>>> x={}                              #空字典
>>> x
{}
>>> type(x)
<type 'dict'>
```

或者使用内置函数 dict()根据给定的“键-值对”来创建字典：

```
>>> d=dict(name='Dong', age=37)
>>> d
{'age': 37, 'name': 'Dong'}
```

还可以以给定内容为“键”，创建“值”为空的字典：

```
>>> adict=dict.fromkeys(['name', 'age', 'sex'])
>>> adict
{'age': None, 'name': None, 'sex': None}
```

当不再需要某个字典时，可以使用 del 命令删除整个字典，也可以使用 del 命令删除字典中指定的元素，请参考 2.3.3 节的内容。

### 2.3.2 字典元素的读取

与列表和元组类似，可以使用下标的方式来访问字典中的元素，但不同的是字典的下标是字典的“键”，而列表和元组访问时下标必须为整数值。使用下标的方式访问字典“值”时，若指定的“键”不存在则抛出异常。

```
>>> aDict={'name':'Dong', 'sex':'male', 'age':37}
>>> aDict['name']
```

```
'Dong'
>>> aDict['tel']
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in<module>
    aDict['tel']
KeyError: 'tel'
```

比较推荐的也是更加安全的字典元素访问方式是字典对象的 get()方法。使用字典对象的 get()方法可以获取指定“键”对应的“值”,并且可以在指定“键”不存在的时候返回指定值,如果不指定,则默认返回 None。

```
>>> print(aDict.get('address'))
None
>>> print(aDict.get('address', 'SDIBT'))
SDIBT
>>> aDict['score']=aDict.get('score', [])
>>> aDict['score'].append(98)
>>> aDict['score'].append(97)
>>> aDict
{'age': 37, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
```

另外,使用字典对象的 items()方法可以返回字典的“键-值”对列表,使用字典对象的 keys()方法可以返回字典的“键”列表,使用字典对象的 values()方法可以返回字典的“值”列表。

```
>>> aDict={'name':'Dong', 'sex':'male', 'age':37}
>>> for item in aDict.items():
    print(item)
('age', 37)
('name', 'Dong')
('sex', 'male')
>>> for key in aDict:
    print(key)
age
name
sex
>>> for key, value in aDict.items():
    print(key, value)
age 37
name Dong
sex male
>>> print(aDict.keys())
['age', 'name', 'sex']
>>> aDict.values()
[37, 'Dong', 'male']
```

### 2.3.3 字典元素的添加与修改

当以指定“键”为下标为字典元素赋值时，若该“键”存在，则表示修改该“键”的值；若不存在，则表示添加一个新的“键-值对”，也就是添加一个新元素。

```
>>> aDict['age']=38
>>> aDict
{'age': 38, 'name': 'Dong', 'sex': 'male'}
>>> aDict['address']='SDIBT'
>>> aDict
{'age': 38, 'address': 'SDIBT', 'name': 'Dong', 'sex': 'male'}
```

使用字典对象的 update()方法将另一个字典的“键-值对”一次性全部添加到当前字典对象，如果两个字典中存在相同的“键”，则以另一个字典中的“值”为准对当前字典进行更新。

```
>>> aDict
{'age': 37, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> aDict.items()
[('age', 37), ('score', [98, 97]), ('name', 'Dong'), ('sex', 'male')]
>>> aDict.update({'a':'a', 'b':'b'})
>>> aDict
{'a': 'a', 'score': [98, 97], 'name': 'Dong', 'age': 37, 'b': 'b', 'sex': 'male'}
```

当需要删除字典元素时，可以根据具体要求使用 del 命令删除字典中指定“键”对应的元素，或者也可以使用字典对象的 clear()方法来删除字典中所有元素，还可以使用字典对象的 pop()方法删除并返回指定“键”的元素，或者使用字典对象的 popitem()方法删除并返回字典中的一个元素，大家可以自行练习这些用法。

### 2.3.4 字典应用案例

下面的代码首先生成包含 1000 个随机字符的字符串，然后统计每个字符的出现次数。

```
>>> import string
>>> import random
>>> x=string.ascii_letters+string.digits+string.punctuation
>>> x
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!"#$%&\'()
*+,-./:;<=>?@[\\]^_'{|}~'
>>> y=[random.choice(x) for i in range(1000)]
>>> z=''.join(y)
>>> d=dict()
```

```
>>> for ch in z:
    d[ch]=d.get(ch, 0)+1
```

也可以使用 collections 模块的 defaultdict 类来实现该功能。

```
>>> import string
>>> import random
>>> x=string.ascii_letters+string.digits+string.punctuation
>>> y=[random.choice(x) for i in range(1000)]
>>> z=''.join(y)
>>> from collections import defaultdict
>>> frequences=defaultdict(int)
>>> frequences
defaultdict(<type 'int'>, {})
>>> for item in z:
    frequences[item]+=1
>>> frequences.items()
```

使用 collections 模块的 Counter 类可以快速实现这个功能，并且能够满足其他需要，例如查找出现次数最多的元素。下面的代码演示了 Counter 类的用法：

```
>>> from collections import Counter
>>> frequences=Counter(z)
>>> frequences.items()
>>> frequences.most_common(1)
[('A', 22)]
>>> frequences.most_common(3)
[('A', 22), (';', 18), ('`', 17)]
```

### 2.3.5 有序字典

Python 内置字典是无序的，前面的示例很好地说明了这个问题。如果需要一个可以记住元素插入顺序的字典，可以使用 collections. OrderedDict。例如下面的代码：

```
>>> x=dict()                          #无序字典
>>> x['a']=3
>>> x['b']=5
>>> x['c']=8
>>> x
{'b': 5, 'c': 8, 'a': 3}
>>> import collections
>>> x=collections.OrderedDict()    #有序字典
>>> x['a']=3
>>> x['b']=5
>>> x['c']=8
```

```
>>> x
OrderedDict([('a', 3), ('b', 5), ('c', 8)])
```

## 2.4 集合

集合是无序可变集合,与字典一样使用一对大括号作为界定符,同一个集合的元素之间不允许重复,集合中每个元素都是唯一的。

### 2.4.1 集合的创建与删除

正如前面多次提到的,在 Python 中变量不需要提前声明其类型,直接将集合赋值给变量即可创建一个集合对象。

```
>>> a={3, 5}
>>> a.add(7)
>>> a
set([3, 5, 7])
```

也可以使用 set()函数将列表、元组等其他可迭代对象转换为集合,如果原来的数据中存在重复元素,则在转换为集合的时候只保留一个。

```
>>> a_set=set(range(8, 14))
>>> a_set
set([8, 9, 10, 11, 12, 13])
>>> b_set=set([0, 1, 2, 3, 0, 1, 2, 3, 7, 8])
>>> b_set
set([0, 1, 2, 3, 7, 8])
>>> x=set()                              #空集合
>>> x
set([])
```

当不再使用某个集合时,可以使用 del 命令删除整个集合。另外,也可以使用集合对象的 pop()方法弹出并删除其中一个元素,或者使用集合对象的 remove()方法直接删除指定元素,以及使用集合对象的 clear()方法清空集合删除所有元素。

```
>>> a={1, 4, 2, 3}
>>> a.pop()
1
>>> a
set([2, 3, 4])
>>> a.pop()
2
>>> a
set([3, 4])
```

```
>>> a.add(2)
>>> a
set([2, 3, 4])
>>> a.remove(3)                #删除指定元素
>>> a
set([2, 4])
>>> a.pop(2)                   #pop()方法不接收参数
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in<module>
    a.pop(2)
TypeError: pop() takes no arguments (1 given)
```

### 2.4.2 集合操作

Python 集合支持交集、并集、差集等运算，大家结合在其他课程里学过的集合知识，应该不难理解下面的代码：

```
>>> a_set=set([8, 9, 10, 11, 12, 13])
>>> b_set=set([0, 1, 2, 3, 7, 8])
>>> a_set | b_set              #并集
set([0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13])
>>> a_set.union(b_set)         #并集
set([0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13])
>>> a_set & b_set              #交集
set([8])
>>> a_set.intersection(b_set) #交集
set([8])
>>> a_set.difference(b_set)    #差集
set([9, 10, 11, 12, 13])
>>> a_set-b_set
set([9, 10, 11, 12, 13])
>>> a_set.symmetric_difference(b_set)        #对称差
set([0, 1, 2, 3, 7, 9, 10, 11, 12, 13])
>>> a_set ^ b_set
set([0, 1, 2, 3, 7, 9, 10, 11, 12, 13])
>>> x={1, 2, 3}
>>> y={1, 2, 5}
>>> z={1, 2, 3, 4}
>>> x<y                        #比较集合大小
False
>>> x<z
True
>>> y<z
```

```
False
>>> x.issubset(y)                #测试是否为子集
False
>>> x.issubset(z)
True
```

作为集合的具体应用，可以使用集合快速提取序列中单一元素，即提取出序列中所有不重复元素，如果使用传统方式，则需要编写下面的代码：

```
>>> import random
>>> listRandom=[random.choice(range(10000)) for i in range(100)]
                                 #100 个介于 0 到 9999 之间的随机数
>>> noRepeat=[]
>>> for i in listRandom :
    if i not in noRepeat :
        noRepeat.append(i)
>>> len(listRandom)
>>> len(noRepeat)
```

如果使用集合的话，只需要如下一行代码就可以了，可以参考上面的代码对结果进行验证。

```
>>> newSet=set(listRandom)
```

## 2.5 再谈内置方法 sorted()

前面已经介绍过，列表对象提供了 sort()方法支持原地排序，而内置函数 sorted()返回新的列表，并不对原列表进行任何修改。除此之外，sorted()方法还可以对元组、字典进行排序，并且借助于其 key 和 cmp 参数(Python 3. x 的 sorted()方法没有 cmp 参数)可以实现更加复杂的排序。需要注意的是，Python 2. x 中内置方法 sorted()的 cmp 参数会被处理多次，而 key 参数只会被处理一次，具有更高的速度。

```
>>> persons=[{'name':'Dong', 'age':37}, {'name':'Zhang', 'age':40}, {'name':
'Li', 'age':50}, {'name':'Dong', 'age':43}]
>>> print(persons)
[{'age': 37, 'name': 'Dong'}, {'age': 40, 'name': 'Zhang'}, {'age': 50, 'name':
'Li'}, {'age': 43, 'name': 'Dong'}]
#使用 key 来指定排序依据，先按姓名升序排序，姓名相同的按年龄降序排序
>>> print(sorted(persons, key=lambda x:(x['name'], -x['age'])))
[{'age': 43, 'name': 'Dong'}, {'age': 37, 'name': 'Dong'}, {'age': 50, 'name':
'Li'}, {'age': 40, 'name': 'Zhang'}]

>>> from timeit import Timer
#在 Python 2.7.8 中比较 sorted()方法的 key 参数与 cmp 参数对排序速度的影响
```

```
>>> Timer(stmt='sorted(xs, key=lambda x: x[1])', setup='xs=range(100); xs=
zip(xs, xs); ').timeit(100000)
1.930681312803035
>>> Timer(stmt='sorted(xs, cmp=lambda a, b: cmp(a[1], b[1]))', setup='xs=
range(100);xs=zip(xs, xs); ').timeit(100000)
3.0562786705272416

>>> phonebook={'Linda':'7750', 'Bob':'9345', 'Carol':'5834'}
>>> from operator import itemgetter
>>> sorted(phonebook.items(), key=itemgetter(1)) #按字典中元素值进行排序
[('Carol', '5834'), ('Linda', '7750'), ('Bob', '9345')]
>>> sorted(phonebook.items(), key=itemgetter(0)) #按字典中元素的键进行排序
[('Bob', '9345'), ('Carol', '5834'), ('Linda', '7750')]

>>> gameresult=[['Bob', 95.0, 'A'], ['Alan', 86.0, 'C'], ['Mandy', 83.5, 'A'],
['Rob', 89.3, 'E']]
>>> sorted(gameresult, key=itemgetter(0, 1)) #按姓名升序,姓名相同按分数升序排序
[['Alan', 86.0, 'C'], ['Bob', 95.0, 'A'], ['Mandy', 83.5, 'A'], ['Rob', 89.3, 'E']]
>>> sorted(gameresult, key=itemgetter(1, 0))
                                            #按分数升序,分数相同的按姓名升序排序
[['Mandy', 83.5, 'A'], ['Alan', 86.0, 'C'], ['Rob', 89.3, 'E'], ['Bob', 95.0, 'A']]
>>> sorted(gameresult, key=itemgetter(2, 0))
                                            #按等级升序,等级相同的按姓名升序排序
[['Bob', 95.0, 'A'], ['Mandy', 83.5, 'A'], ['Alan', 86.0, 'C'], ['Rob', 89.3, 'E']]

>>> gameresult=[{'name':'Bob', 'wins':10, 'losses':3, 'rating':75.0},
                {'name':'David', 'wins':3, 'losses':5, 'rating':57.0},
                {'name':'Carol', 'wins':4, 'losses':5, 'rating':57.0},
                {'name':'Patty', 'wins':9, 'losses':3, 'rating':72.8}]
>>> sorted(gameresult, key=itemgetter('wins', 'name'))
#按'wins'升序,该值相同的按'name'升序排序
[{'wins': 3, 'rating': 57.0, 'name': 'David', 'losses': 5}, {'wins': 4, 'rating':
57.0, 'name': 'Carol', 'losses': 5}, {'wins': 9, 'rating': 72.8, 'name': 'Patty',
'losses': 3}, {'wins': 10, 'rating': 75.0, 'name': 'Bob', 'losses': 3}]
#以下代码演示如何根据另外一个列表的值来对当前列表元素进行排序
>>> list1=["what", "I'm", "sorting", "by"]
>>> list2=["something", "else", "to", "sort"]
>>> pairs=zip(list1, list2)
>>> pairs=sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result=[x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

# 2.6 复杂数据结构

在应用开发中，除了 Python 序列等基本数据类型之外，还经常需要使用到其他一些数据结构，例如堆、栈、队列、树、图等等。其中有些结构 Python 本身已经提供了，而有些则需要自己利用 Python 基本序列或其他数据类型来实现。本节内容可以看作是 Python 序列、元组等基本数据结构的扩展，或者 Python 基本数据结构的二次开发。这里假设你对“数据结构”的知识有所了解，因此有些基本概念就不做过多的解释了，如果需要的话，请自行查阅有关资料。当然，你也可以参考本节的思路自己编写代码来实现“数据结构”课程中更加复杂的数据结构。

## 2.6.1 堆

堆是一种重要的数据结构，在进行排序时使用较多，优先队列也是堆结构的一个重要应用。堆是一个二叉树，其中每个父节点的值都小于或等于其所有子节点的值。使用数组或列表来实现堆时，对于所有的 k(下标，从 0 开始)都满足 heap[k]<=heap[2 * k+1] 和 heap[k]<=heap[2 * k+2]，并且整个堆中最小的元素总是位于二叉树的根节点。Python 在 heapq 模块中提供了对堆的支持。下面的代码演示了堆的原理以及 heapq 模块的用法，同时也请注意 random 模块的用法。另外，当堆中没有元素时，进行 heappop() 操作将会抛出异常。

```
>>> import heapq
>>> import random
>>> data=range(10)
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random.choice(data)                #从序列中随机选择一个元素
9
>>> random.choice(data)
1
>>> random.choice(data)
3
>>> random.shuffle(data)               #随机打乱列表中元素的顺序
>>> data
[6, 1, 3, 4, 9, 0, 5, 2, 8, 7]
>>> heap=[]
>>> for n in data:                     #建堆
    heapq.heappush(heap, n)
>>> heap
[0, 2, 1, 4, 7, 3, 5, 6, 8, 9]
>>> heapq.heappush(heap, 0.5)          #新数据入堆
```

```
>>> heap
[0, 0.5, 1, 4, 2, 3, 5, 6, 8, 9, 7]
>>> heapq.heappop(heap)                  #弹出最小的元素,堆会自动进行重建
0
>>> heapq.heappop(heap)
0.5
>>> heapq.heappop(heap)
1
>>> myheap=[1, 2, 3, 5, 7, 8, 9, 4, 10, 333]
>>> heapq.heapify(myheap)                #将列表转化为堆
>>> myheap
[1, 2, 3, 4, 7, 8, 9, 5, 10, 333]
>>> heapq.heapreplace(myheap, 6)         #替换堆中的元素值,自动重新构建堆
1
>>> myheap
[2, 4, 3, 5, 7, 8, 9, 6, 10, 333]
>>> heapq.nlargest(3, myheap)            #返回前 3 个最大的元素
[333, 10, 9]
>>> heapq.nsmallest(3, myheap)           #返回前 3 个最小的元素
[2, 3, 4]
```

## 2.6.2 队列

队列的特点是“先进先出(First In First Out,FIFO)”和“后进后出(Last In Last Out,LILO)”,在某些应用中有着重要的作用,例如多线程编程、作业处理等等。Python 提供了 Queue 模块(在 Python 3.x 中为 queue)和 collections.deque 模块支持队列的操作,当然也可以使用 Python 列表进行二次开发来实现自定义的队列结构。例如,下面的 Python 2.7.8 代码演示了 Queue 模块的用法:

```
>>> import Queue                         #queue in Python 3.x
>>> q=Queue.Queue()
>>> q.put(0)                             #元素入队,添加到队列尾部
>>> q.put(1)
>>> q.put(2)
>>> print q.queue
deque([0, 1, 2])
>>> print q.get()                        #队列头元素出队
0
>>> print q.queue
deque([1, 2])
>>> print q.get()
1
>>> print q.queue
```

```
deque([2])
```

另外,Queue 和 queue 模块还提供了"后进先出"队列和优先级队列,例如,下面的 Python 3.4.2 代码:

```
>>> import queue
>>> LiFoQueue=queue.LifoQueue(5)        #"后进先出"队列
>>> LiFoQueue.put(1)
>>> LiFoQueue.put(2)
>>> LiFoQueue.put(3)
>>> LiFoQueue.queue
[1, 2, 3]
>>> LiFoQueue.get()
3
>>> LiFoQueue.get()
2
>>> LiFoQueue.get()
1
>>> import queue
>>> PriQueue=queue.PriorityQueue(5)   #优先级队列
>>> PriQueue.put(3)
>>> PriQueue.queue
[3]
>>> PriQueue.put(5)
>>> PriQueue.queue
[3, 5]
>>> PriQueue.put(1)
>>> PriQueue.queue
[1, 5, 3]
>>> PriQueue.put(8)
>>> PriQueue.queue
[1, 5, 3, 8]
>>> PriQueue.get()
1
>>> PriQueue.get()
3
>>> PriQueue.get()
5
>>> PriQueue.get()
8
```

下面的代码使用了 collections 模块中的双端队列。

```
>>> from collections import deque
>>> queue=deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")
```

```
>>> queue.append("Graham")
>>> queue.popleft()
'Eric'
>>> queue.popleft()
'John'
>>> queue
deque(['Michael', 'Terry', 'Graham'])
```

下面的 Python 2.7.8 代码定义了一个类，利用 Python 列表实现了自定义的队列结构，并模拟了队列的基本操作，关于面向对象的知识可以参考第 6 章的介绍。

```
class myQueue:

    def __init__(self, size=10):
        self._content=[]
        self._size=size

    def setSize(self, size):
        self._size=size

    def put(self, v):
        if len(self._content)<self._size:
            self._content.append(v)
        else:
            print 'The queue is full'

    def get(self):
        if self._content:
            return self._content.pop(0)
        else:
            print 'The queue is empty'

    def show(self):
        if self._content:
            print self._content
        else:
            print 'The queue is empty'

    def empty(self):
        self._content=[]

    def isEmpty(self):
        if not self._content:
            return True
        else:
```

```
                return False

        def isFull(self):
            if len(self._content)==self._size:
                return True
            else:
                return False
```

可以使用该类中的方法来实现队列的基本操作，当然也可以对上面的代码进行适当的扩展以实现其他特殊需求。下面的代码简单演示了这个自定义队列结构的用法。

```
>>> import myQueue
>>> q=myQueue.myQueue()
>>> q.get()              #元素出队，队列为空时给出提示
The queue is empty
>>> q.put(5)             #元素入队
>>> q.show()
[5]
>>> q.put(7)
>>> q.put('a')
>>> q.show()
[5, 7, 'a']
>>> q.isEmpty()          #测试队列是否为空
False
>>> q.isFull()           #测试队列是否已满
False
>>> q.get()
5
>>> q.get()
7
>>> q.get()
'a'
>>> q.get()
The queue is empty
```

### 2.6.3 栈

栈是一种“后进先出（Last In First Out，LIFO）”或“先进后出（First In Last Out，FILO）”的数据结构，Python 列表本身就可以实现栈结构的基本操作。例如，列表对象的append()方法是在列表尾部追加元素，类似于入栈操作；pop()方法默认是弹出并返回列表的最后一个元素，类似于出栈操作。但是直接使用 Python 列表对象模拟栈操作并不是很方便，例如，当列表为空时，若再执行 pop()出栈操作，则会抛出一个不很友好的异常；另外，也无法限制栈的大小。例如下面的代码：

```
>>> myStack=[]
>>> myStack.append(3)
>>> myStack.append(5)
>>> myStack.append(7)
>>> myStack
[3, 5, 7]
>>> myStack.pop()
7
>>> myStack.pop()
5
>>> myStack.pop()
3
>>> myStack.pop()
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in<module>
    myStack.pop()
IndexError: pop from empty list
```

下面的 Python 2.7.8 代码使用列表实现了自定义的栈结构来模拟栈结构及其基本操作，除了支持常规的入栈和出栈操作，还有效地控制了栈的大小，并且支持置空和测试栈是否为空、是否为满等状态以及实时查看剩余可用空间。

```
class Stack:

    def __init__(self, size=10):
        self._content=[]
        self._size=size

    def empty(self):
        self._content=[]

    def isEmpty(self):
        if not self._content:
            return True
        else:
            return False

    def setSize(self, size):
        self._size=size

    def isFull(self):
        if len(self._content)==self._size:
            return True
        else:
            return False
```

```
    def push(self, v):
        if len(self._content)<self._size:
            self._content.append(v)
        else:
            print 'Stack Full!'

    def pop(self):
        if self._content:
            return self._content.pop()
        else:
            print 'Stack is empty!'

    def show(self):
        print self._content

    def showRemainderSpace(self):
        print 'Stack can still PUSH ', self._size-len(self._content),
' elements.'

if __name__=='__main__':
    print 'Please use me as a module.'
```

将上面代码保存为 Stack.py 文件之后，可以作为模块进行使用来模拟栈的基本操作。当然，也可以在上面代码的基础上进行扩展以实现你的想法。

```
>>> import Stack
>>> x=Stack.Stack()
>>> x.push(1)
>>> x.push(2)
>>> x.show()
[1, 2]
>>> x.pop()
2
>>> x.show()
[1]
>>> x.showRemainderSpace()
Stack can still PUSH  9  elements.
>>> x.isEmpty()
False
>>> x.isFull()
False
```

### 2.6.4 链表

可以直接使用 Python 列表及其基本操作来实现链表的功能，可以很方便地实现链

表创建以及节点的插入和删除操作，当然也可以对列表进行封装来实现自定义的链表结构实现特殊功能或更加完美的外围检查工作。下面的代码直接使用 Python 列表模拟了链表及其基本操作：

```
>>> linkTable=[]
>>> linkTable.append(3)                 #在尾部追加节点
>>> linkTable.append(5)
>>> linkTable
[3, 5]
>>> linkTable.insert(1, 4)              #在链表中间插入节点
>>> linkTable
[3, 4, 5]
>>> linkTable.remove(linkTable[1])      #删除节点
>>> linkTable
[3, 5]
```

如前所述，使用列表直接模拟链表结构时，同样存在一些问题，例如，链表为空或删除的元素不存在时会抛出异常，可以对列表进行封装来实现完整的链表操作，可以参考队列与栈的代码，此处不再赘述。

### 2.6.5 二叉树

如果学过数据结构，大家肯定还记得，用 C/C++ 来实现二叉树要考虑很多语言本身的问题，看到下面使用 Python 列表实现的二叉树，相信你会眼前一亮。使用代码中的类 BinaryTree 创建的对象不仅支持二叉树的创建以及前序遍历、中序遍历与后序遍历三种常用的二叉树节点遍历方式，还支持二叉树中任意“子树”的遍历。

```
#-*-coding:utf-8-*-
#Filename: BinaryTree.py
#--------------------
#Function description:
#Creation, traverse of binary tree
#--------------------
#Author: Dong Fuguo
#QQ: 306467355
#Email: dongfuguo2005@126.com
#--------------------
#Date: 2014-11-15
#--------------------
class BinaryTree:

    def __init__(self, value):
        self.__left=None
```

```
        self.__right=None
        self.__data=value

    def insertLeftChild(self, value):  #插入左节点
        if self.__left:
            print '__left child tree already exists.'
        else:
            self.__left=BinaryTree(value)
            return self.__left

    def insertRightChild(self, value): #插入右节点
        if self.__right:
            print 'Right child tree already exists.'
        else:
            self.__right=BinaryTree(value)
            return self.__right

    def removeLeftChild(self):         #删除左子树
        self.__left=None

    def removeRightChild(self):        #删除右子树
        self.__right=None

    def show(self):
        print self.__data

    def preOrder(self):                #前序遍历
        print self.__data
        if self.__left:
            self.__left.preOrder()
        if self.__right:
            self.__right.preOrder()

    def postOrder(self):               #后序遍历
        if self.__left:
            self.__left.postOrder()
        if self.__right:
            self.__right.postOrder()
        print self.__data

    def inOrder(self):                 #中序遍历
        if self.__left:
            self.__left.inOrder()
        print self.__data
```

```
        if self.__right:
            self.__right.inOrder()

if __name__=='__main__':
    print 'Please use me as a module.'
```

这段代码可以不进行任何修改地运行于 Python 2. x，如果使用 Python 3. x 版本，仅需要把代码中的 print 语句改为 print()函数即可，前面有过类似的说明，此处不再赘述。假设把上面的代码保存为文件 BinaryTree. py，然后使用下面的方法来使用上面定义的二叉树类。需要把这个文件放在 Python 的安装目录中，或者把含有该文件的目录添加到 sys. path 中。

```
>>> import BinaryTree
>>> root=BinaryTree.BinaryTree('root')
>>> firstRight=root.insertRightChild('B')
>>> firstLeft=root.insertLeftChild('A')
>>> secondLeft=firstLeft.insertLeftChild('C')
>>> thridRight=secondLeft.insertRightChild('D')
>>> root.postOrder()                        #后序遍历
D
C
A
B
root
>>> root.preOrder()                         #前序遍历
root
A
C
D
B
>>> root.inOrder()                          #中序遍历
C
D
A
root
B
>>> firstLeft.inOrder()                     #遍历“子树”
C
D
A
>>> secondLeft.removeRightChild()           #删除二叉树中的节点
>>> root.preOrder()
root
A
```

```
C
B
```

### 2.6.6 有向图

作为本章的最后一个示例，让我们来看一下使用 Python 语言实现有向图的创建和路径搜索。有向图由节点和边组成，而每条边都是有方向的，若两个节点之间存在有向边，则表示可以从起点到达终点。与二叉树的示例一样，下面给出较为完整的代码。

```
#-*-coding:utf-8-*-
#Filename: DirectedGraph.py
#--------------------
#Function description:path searching of directed graph
#--------------------
def searchPath(graph, start, end):
    results=[]
    __generatePath(graph, [start], end, results)
    results.sort(key=lambda x: len(x))
return results

def __generatePath(graph, path, end, results):
    current=path[-1]
    if current==end:
        results.append(path)
    else:
        for n in graph[current]:
            if n not in path:
                #path.append(n)
                __generatePath(graph, path+[n], end, results)

def showPath(results):
    print 'The path from ', results[0][0], ' to ', results[0][-1], ' is:'
    for path in results:
        print path

if __name__=='__main__':
    graph={'A':['B', 'C', 'D'],
           'B':['E'],
           'C':['D', 'F'],
           'D':['B', 'E', 'G'],
           'E':['G'],
           'F':['D', 'G'],
           'G':['E', 'A', 'B']}
```

```
r=searchPath(graph, 'A', 'D')
showPath(r)
```

读者可以自行运行该程序,并结合运行结果来理解这段代码。

## 本章小结

(1) 列表、字符串、元组属于有序序列,支持双向索引,支持使用负整数作为下标来访问其中的元素,-1 表示最后一个元素位置,-2 表示倒数第二个元素位置,以此类推。

(2) 在 Python 中,同一个列表中元素的数据类型可以各不相同,可以同时分别为整数、实数、字符串等基本类型,也可以是列表、元组、字典、集合以及其他自定义类型的对象,并且支持复杂数据类型对象的嵌套。

(3) 字典和集合属于无序序列,集合不支持使用下标的方式来访问其中的元素,可以使用字典的"键"作为下标来访问字典中的"值"。

(4) 如果要创建只包含一个元素的元组,只把元素放在圆括号里是不行的,还需要在元素后面加一个逗号","。

(5) 将列表、元组或字符串对象与一个整数进行"*"运算,表示将对象中的元素进行重复并返回一个新的同类型对象。

(6) 虽然"+"运算符可以连接两个列表对象,但并不是原地修改列表,而是返回一个新列表,不对原列表对象做任何修改。并且该运算符涉及到大量的元素赋值操作,效率较低,建议优先考虑使用列表对象的 append()方法。

(7) 推荐使用字典对象的 get()来访问其中的元素。

(8) 列表、字典、集合属于可变序列,元组、字符串属于不可变序列。

(9) 虽然列表支持在列表中间任意位置插入和删除元素,但一般建议尽量从列表的尾部进行元素的增加与删除,这样可以获得更高的速度。

(10) 切片操作不仅可以用来返回列表、元组、字符串中的部分元素,还可以对列表中的元素值进行修改,以及增加或删除列表中的元素。

(11) 关键字 in 可以用于列表以及其他可迭代对象,包括元组、字典、range 对象、字符串、集合等等,常用在循环语句中对序列或其他可迭代对象中的元素进行遍历。

(12) 列表推导式可以使用简洁的形式来生成满足特定需要的列表。

(13) 序列解包在多个场合具有重要的应用,是 Python 的基本操作之一。

(14) 字典中的"键"可以是 Python 中任意不可变数据,比如整数、实数、复数、字符串、元组等等,但不能使用列表、集合、字典作为字典的"键",因为这些类型的对象是可变的。

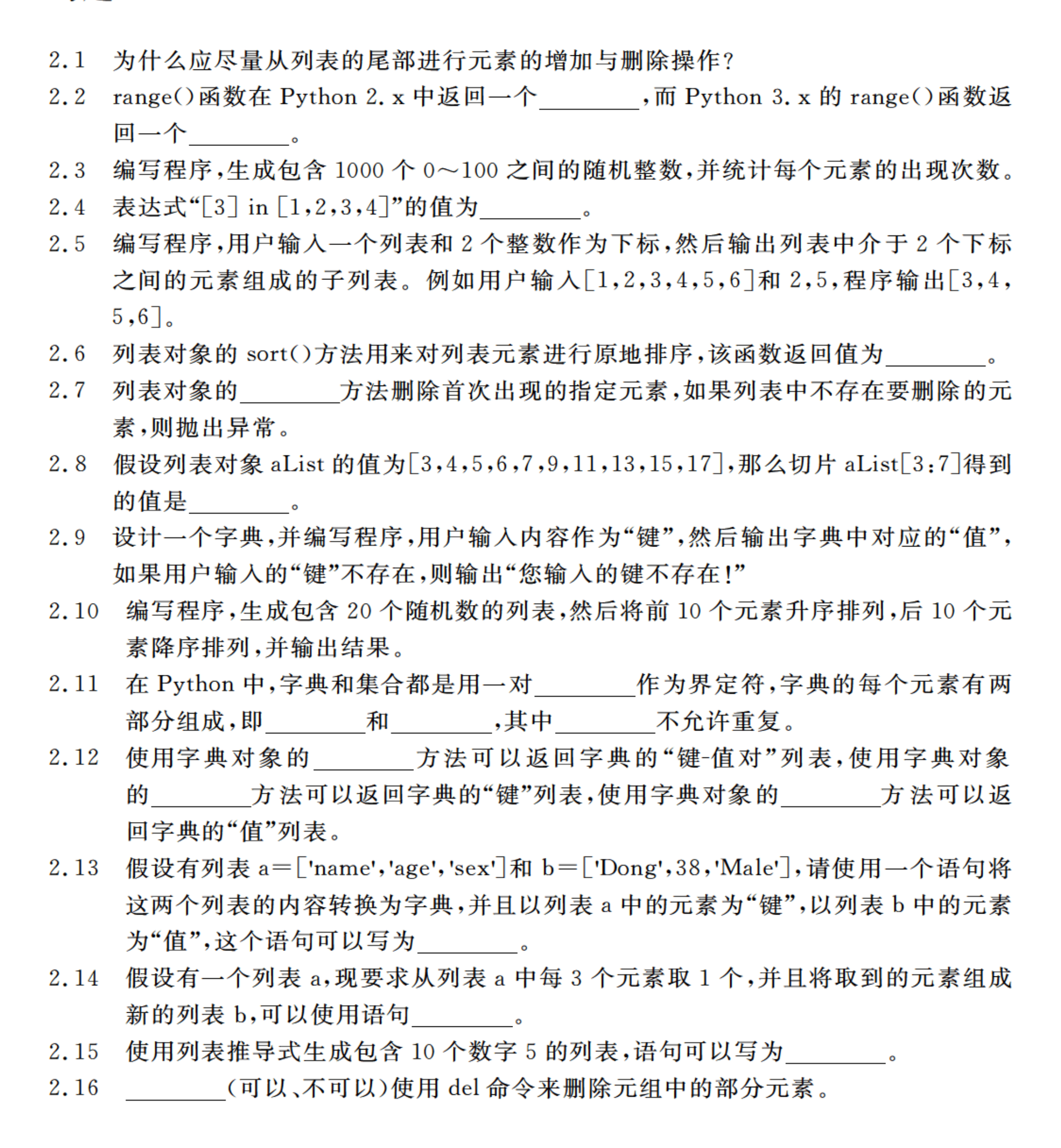
(15) 字典中的"键"不允许重复,"值"是可以重复的。

(16) 集合中的所有元素不允许重复,可以使用集合快速提取其他序列中的唯一元素。

(17) 内置函数 len(列表)可以用来返回列表中的元素个数,同样适用于元组、字典、集合、字符串、range 对象等其他可迭代对象。

(18) 内置函数 zip(列表 1,列表 2,…)可以将多个列表或元组对应位置的元素组合为元组,并返回包含这些元组的列表(Python 2.x)或 zip 对象(Python 3.x)。

(19) 内置函数 enumerate(可迭代对象)可以用来枚举列表、元组或其他可迭代对象的元素,返回枚举对象,枚举对象中每个元素是包含下标和元素值的元组。



## 习题

2.1 为什么应尽量从列表的尾部进行元素的增加与删除操作?

2.2 range()函数在 Python 2.x 中返回一个________,而 Python 3.x 的 range()函数返回一个________。

2.3 编写程序,生成包含 1000 个 0～100 之间的随机整数,并统计每个元素的出现次数。

2.4 表达式“[3] in [1,2,3,4]”的值为________。

2.5 编写程序,用户输入一个列表和 2 个整数作为下标,然后输出列表中介于 2 个下标之间的元素组成的子列表。例如用户输入[1,2,3,4,5,6]和 2,5,程序输出[3,4,5,6]。

2.6 列表对象的 sort()方法用来对列表元素进行原地排序,该函数返回值为________。

2.7 列表对象的________方法删除首次出现的指定元素,如果列表中不存在要删除的元素,则抛出异常。

2.8 假设列表对象 aList 的值为[3,4,5,6,7,9,11,13,15,17],那么切片 aList[3:7]得到的值是________。

2.9 设计一个字典,并编写程序,用户输入内容作为“键”,然后输出字典中对应的“值”,如果用户输入的“键”不存在,则输出“您输入的键不存在!”

2.10 编写程序,生成包含 20 个随机数的列表,然后将前 10 个元素升序排列,后 10 个元素降序排列,并输出结果。

2.11 在 Python 中,字典和集合都是用一对________作为界定符,字典的每个元素有两部分组成,即________和________,其中________不允许重复。

2.12 使用字典对象的________方法可以返回字典的“键-值对”列表,使用字典对象的________方法可以返回字典的“键”列表,使用字典对象的________方法可以返回字典的“值”列表。

2.13 假设有列表 a=['name','age','sex']和 b=['Dong',38,'Male'],请使用一个语句将这两个列表的内容转换为字典,并且以列表 a 中的元素为“键”,以列表 b 中的元素为“值”,这个语句可以写为________。

2.14 假设有一个列表 a,现要求从列表 a 中每 3 个元素取 1 个,并且将取到的元素组成新的列表 b,可以使用语句________。

2.15 使用列表推导式生成包含 10 个数字 5 的列表,语句可以写为________。

2.16 ________(可以、不可以)使用 del 命令来删除元组中的部分元素。

# 第 3 章　选择与循环

在传统的面向过程程序设计中有三种经典的控制结构，即顺序结构、选择结构和循环结构。即使是在面向对象程序设计语言中以及事件驱动或消息驱动应用开发中，也无法脱离这三种基本的程序结构。可以说，不管使用哪种程序设计语言，在实际开发中，为了实现特定的业务逻辑或算法，都不可避免地要用到大量的选择结构和循环结构，并且经常需要将选择结构和循环结构嵌套使用。本章首先介绍条件表达式和 Python 中选择结构与循环结构的语法，然后通过几个示例来理解其用法。

## 3.1　条件表达式

在选择结构和循环结构中，都要使用条件表达式来确定下一步的执行流程。在 Python 中，单个常量、变量或者任意合法表达式都可以作为条件表达式。在条件表达式中可以使用 1.4.5 节介绍的所有运算符。

- 算术运算符：＋、－、*、/、//、%、**
- 关系运算符：＞、＜、＝＝、＜＝、＞＝、!＝(Python 2.x 还支持“＜＞”运算符表示不等于，Python 3.x 不支持“＜＞”运算符)
- 测试运算符：in、not in、is、is not
- 逻辑运算符：and、or、not
- 位运算符：～、&、|、^、＜＜、＞＞

在选择和循环结构中，条件表达式的值只要不是 False、0(或 0.0、0j 等)、空值 None、空列表、空元组、空集合、空字典、空字符串、空 range 对象或其他空迭代对象，Python 解释器均认为与 True 等价。从这个意义上来讲，几乎所有的 Python 合法表达式都可以作为条件表达式，包括含有函数调用的表达式。例如：

```
>>> if 3:                #使用整数作为条件表达式
    print(5)
5
>>> a=[1, 2, 3]
>>> if a:                #使用列表作为条件表达式
    print(a)
[1, 2, 3]
>>> a=[]
>>> if a:
    print(a)
else:
    print('empty')
```

```
empty
>>> i=s=0
>>> while i<=10:     #使用关系表达式作为条件表达式
    s+=i
    i+=1
>>> print(s)
55
>>> i=s=0
>>> while True:      #使用常量 True 作为条件表达式
    s+=i
    i+=1
    if i>10:
        break
>>> print(s)
55
>>> s=0
>>> for i in range(0, 11, 1):
    s+=i
>>> print(s)
55
```

关于表达式和运算符的详细内容在 1.4.5 节中已有介绍，此处不再赘述，只简单介绍一下条件表达式中比较特殊的几个运算符。首先是关系运算符，与很多语言不同的是，在 Python 中的关系运算符可以连续使用，如

```
>>> print(1<2<3)
True
>>> print(1<2>3)
False
>>> print(1<3>2)
True
```

比较特殊的运算符还有逻辑运算符 and 和 or，这两个运算符具有短路求值或惰性求值的特点，简单地说，就是只计算必须计算的表达式的值。在设计条件表达式时，在表示复杂条件时如果能够巧妙利用逻辑运算符 and 和 or 的短路求值或惰性求值特性，可以大幅度提高程序的运行效率，减少不必要的计算与判断。以 and 为例，对于表达式“表达式 1 and 表达式 2”而言，如果“表达式 1”的值为 False 或其他等价值时，不论“表达式 2”的值是什么，整个表达式的值都是 False，此时“表达式 2”的值无论是什么都不影响整个表达式的值，因此将不会被计算，从而减少不必要的计算和判断。逻辑或运算符 or 也具有类似的特点，读者可以自行分析。在设计条件表达式时，如果能够大概预测不同条件失败的概率，并将多个条件根据 and 和 or 运算的短路求值特性进行组织，可以大幅度提高程序运行效率。例如，下面的函数用来使用用户指定的分隔符将多个字符串连接成一个字符

串,如果用户没有指定分隔符则使用逗号。

```
>>> def Join(chList, sep=None):
    return (sep or ',').join(chList)
>>> chTest=['1', '2', '3', '4', '5']
>>> Join(chTest)
'1,2,3,4,5'
>>> Join(chTest, ':')
'1:2:3:4:5'
>>> Join(chTest, ' ')
'1 2 3 4 5'
```

当然,还可以把上面的函数直接定义为下面的形式:

```
>>> def Join(chList, sep=','):
    return sep.join(chList)
```

另外,在 Python 中,条件表达式中不允许使用赋值运算符"=",避免了其他语言中误将关系运算符"=="写作赋值运算符"="带来的麻烦,例如下面的代码,在条件表达式中使用赋值运算符"="将抛出异常,提示语法错误。

```
>>> if a=3:
SyntaxError: invalid syntax
>>> if (a=3) and (b=4):
SyntaxError: invalid syntax
```

## 3.2 选择结构

选择结构通过判断某些特定条件是否满足来决定下一步的执行流程,是非常重要的控制结构。常见的有单分支选择结构、双分支选择结构、多分支选择结构、嵌套的分支结构,形式比较灵活多变,具体使用哪一种最终还是取决于所要实现的业务逻辑。从某种意义上讲,后面章节中讲到的循环结构和异常处理结构中也可以带有 else 子句,也可以看作是选择结构的一种变形。

### 3.2.1 单分支选择结构

单分支选择结构是最简单的一种形式,其语法如下所示,其中表达式后面的冒号":"是不可缺少的,表示一个语句块的开始,后面几种其他形式的选择结构和循环结构中的冒号也是必须要有的。

```
if 表达式:
    语句块
```

当表达式值为 True 或其他等价值时,表示条件满足,语句块将被执行,否则该语句

块将不被执行。

```
a, b=input('Input two numbers:')
if a>b:
   a, b=b, a
print(a, b)
```

### 3.2.2　双分支选择结构

双分支选择结构的语法为：

```
if 表达式:
    语句块 1
else:
    语句块 2
```

当表达式值为 True 或其他等价值时，执行语句块 1，否则执行语句块 2。下面的代码演示了双分支选择结构的用法：

```
>>> chTest=['1', '2', '3', '4', '5']
>>> if chTest:
    print(chTest)
else:
    print('Empty')
['1', '2', '3', '4', '5']
```

Python 还支持如下形式的表达式：

```
value1 if condition else value2
```

当条件表达式 condition 的值与 True 等价时，表达式的值为 value1，否则表达式的值为 value2。另外，在 value1 和 value2 中还可以使用复杂表达式，包括函数调用和基本输出语句。下面的代码演示了上面的表达式的用法，从代码中可以看出，这个结构的表达式也具有惰性求值的特点。

```
>>> a=5
>>> print(6) if a>3 else print(5)
6
>>> print(6 if a>3 else 5)
6
>>> b=6 if a>13 else 9
>>> b
9
>>> x=math.sqrt(9) if 5>3 else random.randint(1, 100) #此时还没有导入 math 模块
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in<module>
```

```
    x=math.sqrt(9) if 5>3 else random.randint(1,100)
NameError: name 'math' is not defined
>>> import math
#此时还没有导入 random 模块,但由于条件表达式 5>3 的值为 True,所以可以正常运行
>>> x=math.sqrt(9) if 5>3 else random.randint(1,100)
#此时还没有导入 random 模块,由于条件表达式 2>3 的值为 False,需要计算第二个表达式的值,因此出错
>>> x=math.sqrt(9) if 2>3 else random.randint(1, 100)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in<module>
    x=math.sqrt(9) if 2>3 else random.randint(1,100)
NameError: name 'random' is not defined
>>> import random
>>> x=math.sqrt(9) if 2>3 else random.randint(1, 100)
```

### 3.2.3 多分支选择结构

多分支选择结构为用户提供了更多的选择,可以实现复杂的业务逻辑,多分支选择结构的语法为:

```
if 表达式 1:
    语句块 1
elif 表达式 2:
    语句块 2
elif 表达式 3:
    语句块 3
    ⋮
else:
    语句块 n
```

其中,关键字 elif 是 else if 的缩写。下面的代码演示了利用多分支选择结构将成绩从百分制变换到等级制的实现方法。

```
>>> def func(score):
    if score>100:
        return 'wrong score.must<=100.'
    elif score>=90:
        return 'A'
    elif score>=80:
        return 'B'
    elif score>=70:
        return 'C'
    elif score>=60:
        return 'D'
```

```
    elif score>=0:
        return 'E'
    else:
        return 'wrong score.must>0'
>>> func(120)
'wrong score.must<=100.'
>>> func(99)
'A'
>>> func(87)
'B'
>>> func(62)
'D'
>>> func(3)
'E'
>>> func(-10)
'wrong score.must>0'
```

### 3.2.4 选择结构的嵌套

选择结构可以进行嵌套,语法如下所示:

```
if 表达式 1:
    语句块 1
    if 表达式 2:
        语句块 2
    else:
        语句块 3
else:
    if 表达式 4:
        语句块 4
```

使用该结构时,一定要严格控制好不同级别代码块的缩进量,因为这决定了不同代码块的从属关系以及业务逻辑是否被正确地实现、是否能够被 Python 正确理解和执行。例如 3.2.3 节中百分制转等级制的示例,作为一种编程技巧,还可以尝试下面的写法:

```
>>> def func(score):
    degree='DCBAAE'
    if score>100 or score<0:
        return 'wrong score.must between 0 and 100.'
    else:
        index=(score-60)//10
        if index>=0:
            return degree[index]
        else:
```

```
            return degree[-1]
>>> func(-10)
'wrong score.must between 0 and 100.'
>>> func(30)
'E'
>>> func(50)
'E'
>>> func(60)
'D'
>>> func(93)
'A'
>>> func(100)
'A'
```

最后,需要注意的是,在 IDLE 交互式环境中,每次只能执行一条语句。因此,如果需要编写多条语句实现复杂的业务逻辑,需要创建一个 Python 程序文件。例如,下面的代码无法在 IDLE 交互式环境中运行,而是抛出异常提示语法错误。

```
>>> if 3>5:
    print('n')
if 4>5:            #在交互式环境中直接输入,此处回车后抛出异常

SyntaxError: invalid syntax
#从其他文本编辑器中提前写好多条语句,然后复制到 IDLE 交互式环境中,抛出异常
>>> 3+5
5+8
9+9

SyntaxError: multiple statements found while compiling a single statement
#从其他文本编辑器中编写两条语句,复制并粘贴到 IDLE 交互式环境中运行,抛出异常
>>> if 3>5:
    print('n')
if 4>5:
    print('y')

SyntaxError: invalid syntax
```

### 3.2.5 选择结构应用案例

**例 3-1** 面试资格确认。

```
age=24
subject="计算机"
college="非重点"
```

```
if (age>25 and subject=="电子信息工程") or (college=="重点" and subject=="电子信息工程" ) or (age<=28 and subject=="计算机"):
    print("恭喜,您已获得我公司的面试机会!")
else:
    print("抱歉,您未达到面试要求")
```

**例 3-2** 用户输入若干个成绩,求所有成绩的总和。每输入一个成绩后询问是否继续输入下一个成绩,回答 yes 就继续输入下一个成绩,回答 no 就停止输入成绩。

```
endFlag='yes'
s=0
while endFlag.lower()==' yes':
    x=input("请输入一个正整数: ")
    x=eval(x)
    if isinstance(x, int) and 0<=x<=100:
        s=s+x
    else:
        print('不是数字或不符合要求')
    endFlag=raw_input('继续输入? (yes or no)')
print('整数之和=', s)
```

**例 3-3** 编写程序,判断今天是今年的第几天。

```
import time
date=time.localtime()
year=date[0]
month=date[1]
day=date[2]
day_month=[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
if year%400==0 or (year%4==0 and year%100!=0):      #判断是否为闰年
    day_month[1]=29
if month==1:
    print(day)
else:
    print(sum(day_month[:month-1])+day)
```

## 3.3 循环结构

### 3.3.1 for 循环与 while 循环

Python 提供了两种基本的循环结构: while 循环和 for 循环。其中,while 循环一般用于循环次数难以提前确定的情况,当然也可以用于循环次数确定的情况;for 循环一般用于循环次数可以提前确定的情况,尤其适用于枚举或遍历序列或迭代对象中元素的场合,编程时一般建议优先考虑使用 for 循环。相同或不同的循环结构之间可以互相嵌套,

也可以与选择结构嵌套使用，用来实现更为复杂的逻辑。

while 循环和 for 循环常见的用法为：

```
while 条件表达式:
    循环体
```

和

```
for 变量 in 序列或其他迭代对象:
    循环体
```

另外，while 循环和 for 循环都可以带 else 子句，如果循环因为条件表达式不成立而自然结束（不是因为执行了 break 而结束循环），则执行 else 结构中的语句；如果循环是因为执行了 break 语句而导致循环提前结束，则不执行 else 中的语句。其语法形式为：

```
while 条件表达式:
    循环体
else:
    else 子句代码块
```

和

```
for 取值 in 序列或迭代对象:
    循环体
else:
    else 子句代码块
```

例如，下面的代码演示了带有 else 子句的循环结构，该代码用来计算 1＋2＋3＋…＋99＋100 的结果。

```
>>> s=0
>>> for i in range(1, 101):
    s+=i
else:
    print(s)
5050
```

下面的代码使用 while 循环实现了同样的功能：

```
>>> s=i=0
>>> while i<=100:
    s+=i
    i+=1
else:
    print(s)
5050
```

### 3.3.2 循环结构的优化

为了优化程序以获得更高的效率和运行速度，在编写循环语句时，应尽量减少循环内部不必要的计算，将与循环变量无关的代码尽可能地提取到循环之外。对于使用多重循环嵌套的情况，应尽量减少内层循环中不必要的计算，尽可能地向外提。例如下面的代码，第二段明显比第一段的运行效率要高。

```
import time
digits=(1, 2, 3, 4)

start=time.time()
for i in range(1000):
    result=[]
    for i in digits:
        for j in digits:
            for k in digits:
                result.append(i*100+j*10+k)
print(time.time()-start)
print(result)

start=time.time()
for i in range(1000):
    result=[]
    for i in digits:
        i=i*100
        for j in digits:
            j=j*10
            for k in digits:
                result.append(i+j+k)
print(time.time()-start)
print(result)
```

运行结果如下（为节约篇幅，省略了 result 列表元素）：

```
0.03800201416015625
0.022001028060913086
```

另外，在循环中应尽量引用局部变量，因为局部变量的查询和访问速度比全局变量略快，在使用模块中的方法时，可以通过将其转换为局部变量来提高运行速度。例如下面的代码：

```
import time
import math
```

```
start=time.time()                        #获取当前时间
for i in xrange(10000000):
    math.sin(i)
print('Time Used:', time.time()-start)  #输出所用时间
loc_sin=math.sin

start=time.time()
for i in xrange(10000000):
    loc_sin(i)
print('Time Used:', time.time()-start)
```

这段代码演示了模块方法的两种不同调用方式，并比较各自的运行时间，结果为：

```
('Time Used:', 4.9059998989105225)
('Time Used:', 4.406000137329102)
```

第 1 章还介绍过另外一种导入和使用模块成员的方法，把上面的代码修改为：

```
import time
from math import sin as sin

start=time.time()
for i in xrange(10000000):
    sin(i)
print('Time Used:', time.time()-start)

loc_sin=sin
start=time.time()
for i in xrange(10000000):
    loc_sin(i)
print('Time Used:', time.time()-start)
```

代码运行结果如下，可以看出，效率也略有提高。

```
('Time Used:', 4.608999967575073)
('Time Used:', 4.4059998989105225)
```

## 3.4 break 和 continue 语句

break 语句和 continue 语句在 while 循环和 for 循环中都可以使用，并且一般常与选择结构结合使用，以达到在特定条件得到满足时跳出循环的目的。一旦 break 语句被执行，将使得整个循环提前结束。continue 语句的作用是终止本次循环，并忽略 continue 之后的所有语句，直接回到循环的顶端，提前进入下一次循环。需要注意的是，过多的 break 和 continue 会严重降低程序的可读性。除非 break 或 continue 语句可以让代码更简单或更清晰，否则不要轻易使用。

下面的代码用来计算小于 100 的最大素数，请注意 break 语句和 else 子句的用法。

```
>>> for n in range(100, 1, -1):
    for i in range(2, n):
        if n%i==0:
            break
    else:
        print(n)
        break
97
```

删除上面代码中最后一个 break 语句，则可以用来输出 100 以内的所有素数，例如：

```
>>> for n in range(100, 1, -1):
    for i in range(2, n):
        if n%i==0:
            break
    else:
        print(n, end=' ')
97 89 83 79 73 71 67 61 59 53 47 43 41 37 31 29 23 19 17 13 11 7 5 3 2
```

在编写循环结构代码时，一定要警惕 continue 语句可能带来的问题，例如，下面的代码本意是用来输出 10 以内的奇数：

```
>>> i=1
>>> while i<10:
    if i%2==0:
        continue
    print(i, end=' ')
    i+=1
```

但是由于代码设计存在问题，从而导致这个循环变成了永不结束的死循环，需要按 Ctrl+C 组合键来强行终止。出现这种情况的原因是：一旦条件表达式“i%2==0”得到满足以后执行 continue 语句，之后的“i+=1”语句将永远不再执行，循环变量永远停留在当前的值，从而使得循环无法结束。上面的代码改成下面这样就不会有问题了：

```
>>> i=1
>>> while i<10:
    if i%2==0:
        i+=1
        continue
    print(i, end=' ')
    i+=1
1 3 5 7 9
```

或者修改为下面更为简洁易理解的形式：

```
>>> i=0
>>> while i<10:
    i+=1
    if i%2==0:
        continue
    print(i, end=' ')
1 3 5 7 9
```

当然，也可以使用更简洁的 for 循环来实现：

```
>>> for i in range(10):
    if i%2==0:
        continue
    print(i, end=' ')
1 3 5 7 9
```

为了充分理解，让我们再修改一下：

```
>>> for i in range(10):
    if i%2==0:
        i+=1
        continue
    print(i, end=' ')
1 3 5 7 9
```

在这段代码中，条件语句中 continue 之前的语句“i+=1”并没有起到任何作用。之所以会这样，是因为 Python 基于值的内存管理方式。在上面的代码中，每次进入循环时的变量 i 已经不再是上一次的变量 i，所以修改其值并不会影响循环的执行。下面的代码很好地描述了这个问题。

```
>>> for i in range(5):
    print(id(i), ':', i)
10416692 : 0
10416680 : 1
10416668 : 2
10416656 : 3
10416644 : 4
```

## 3.5 案例精选

本章最后通过几个示例来演示选择结构和循环结构的用法，正如前面所说，这两个结构经常需要互相结合来实现特定的业务逻辑。

**例 3-4** 计算 1+2+3+…+100 的值。

对于这样比较规则的循环，一般优先考虑使用 for 循环，参考 Python 2.7.8 代码

如下：

```
s=0
for i in range(1, 101):
    s=s+i
print('1+2+3+...+100=', s)
print('1+2+3+...+100=', sum(range(1, 101)))    #直接使用内置函数来实现题目的要求
```

类似的问题也可以使用 while 循环解决，请参考 3.3 节的代码。

**例 3-5** 输出序列中的元素。

对于类似元素遍历的问题，一般也优先考虑使用 for 循环，参考代码如下：

```
a_list=['a', 'b', 'mpilgrim', 'z', 'example']
for i, v in enumerate(a_list):
    print('列表的第', i+1, '个元素是：', v)
```

对于类似元素遍历的问题，同样也可以使用 while 循环来解决，但是代码要麻烦一些，可读性也较差，例如：

```
>>> a_list=['a', 'b', 'mpilgrim', 'z', 'example']
>>> i=0
>>> number=len(a_list)
>>> while i<number:
    print('列表的第', i+1, '个元素是：', a_list[i])
    i+=1
列表的第 1 个元素是：a
列表的第 2 个元素是：b
列表的第 3 个元素是：mpilgrim
列表的第 4 个元素是：z
列表的第 5 个元素是：example
```

**例 3-6** 求 1～100 之间能被 7 整除，但不能同时被 5 整除的所有整数。

该例主要介绍条件表达式的写法，参考代码如下：

```
for i in range(1, 101):
    if i %7==0 and i %5 !=0:
        print(i)
```

**例 3-7** 输出“水仙花数”。所谓水仙花数是指 1 个 3 位的十进制数，其各位数字的立方和恰好等于该数本身。例如，153 是水仙花数，因为 $153=1^3+5^3+3^3$。

```
for i in range(100, 1000):
    ge=i %10
    shi=i // 10 %10
    bai=i // 100
    if ge**3+shi**3+bai**3==i:
        print(i)
```

**例 3-8** 求平均分。

```
score=[70, 90, 78, 85, 97, 94, 65, 80]
s=0
for i in score:
    s+=i
print(s/len(score))
```

如果熟悉前面学过的序列知识，就会想到，其实可以使用下面的内置函数来计算平均分：

```
print(sum(score)/len(score))
```

上面的代码是 Python 3.4.2 中编写的，如果使用 Python 2.x，输出语句需要写成下面的形式：

```
print sum(score) * 1.0/len(score)
```

这里之所以需要将所有元素的和乘以 1.0 转换为浮点数，是因为在 Python 2.x 中除法运算符“/”的限制。在 Python 3.x 中“/”被解释为真除法，则不再需要这个转换，详见 1.4.5 节关于运算符的讨论。

**例 3-9** 打印九九乘法表。

该例主要介绍循环结构嵌套用法和循环条件的控制，参考代码如下：

```
for i in range(1, 10):
    for j in range(1, i+1):
        print(i, '*', j, '=', i*j, '\t', end=' ')
    print() #打印空行
```

**例 3-10** 求 200 以内能被 17 整除的最大正整数。

熟练掌握 range()函数的用法，对于很多循环来说可能起到事半功倍的效果，参考代码如下：

```
for i in range(200, 0, -1):
    if i%17==0:
        print(i)
        break
```

**例 3-11** 判断一个数是否为素数。

在该例中，重点演示循环结构中 else 子句的用法。

```
import math
n=input('Input an inter:')
n=int(n)
m=math.ceil(math.sqrt(n)+1)
for i in range(2, m):
    if n% i==0 and i<n:
```

```
        print('No')
        break
else:
    print('Yes')
```

**例 3-12** 鸡兔同笼问题。假设共有鸡、兔 30 只，脚 90 只，求鸡、兔各有多少只。

```
for ji in range(0, 31):
    if 2*ji+(30-ji)*4==90:
        print('ji:', ji, ' tu:', 30-ji)
```

**例 3-13** 编写程序，输出由 1、2、3、4 这四个数字组成的每位数都不相同的所有三位数。

```
digits=(1, 2, 3, 4)
for i in digits:
    for j in digits:
        for k in digits:
            if i!=j and j!=k and i!=k:
                print(i*100+j*10+k)
```

从代码优化的角度来讲，上面这段代码并不是很好，其中有些判断完全可以在外层循环来做，从而提高运行效率，即下面形式的代码运行效率比上面的代码要高一些，可以自行测试。

```
digits=(1, 2, 3, 4)
for i in digits:
    for j in digits:
        if j==i:
            continue
        for k in digits:
            if k==i or k==j:
                continue
            print(i*100+j*10+k)
```

**例 3-14** 编写程序，生成一个含有 20 个随机数的列表，要求所有元素不相同，并且每个元素的值介于 1～100 之间。

```
import random

x=[]
while True:
    if len(x)==20:
        break
    n=random.randint(1, 100)
    if n not in x:
```

```
        x.append(n)
print(x)
print(len(x))
print(sorted(x))
```

## 本章小结

(1) 几乎所有合法的 Python 表达式都可以作为选择结构和循环结构中的条件表达式。

(2) Python 的关系运算符可以连续使用,例如,3<4<5>2 的值为 True。

(3) 数字 0、0.0、0j、逻辑假 False、空列表[]、空集合或空字典{}、空元组()、空字符串''、空值 None 以及任意与这些值等价的值作为条件表达式时均被认为条件不成立,否则认为条件表达式成立。

(4) 逻辑运算符 and 和 or 具有短路求值或惰性求值特点,即只计算必须计算的表达式的值。

(5) 选择结构和循环结构往往会互相嵌套使用来实现复杂的业务逻辑。

(6) 关键字 elif 表示 else if 的意思。

(7) 应优先考虑使用 for 循环,尤其是列表、元组、字典或其他 Python 序列元素遍历的场合。

(8) 编写循环语句时,应尽量减少内循环中的无关计算,对循环进行必要的优化。

(9) for 循环和 while 循环都可以带有 else 子句,如果循环因为条件表达式不满足而自然结束时,执行 else 子句中的代码;如果循环是因为执行了 break 语句而结束,则不执行 else 子句中的代码。

(10) break 语句用来提前结束其所在循环,continue 语句用来提前结束本次循环并进入下一次循环。

(11) 除非 break 和 continue 语句可以让代码变得更简单或更清晰,否则请不要轻易使用。

## 习题

3.1 分析逻辑运算符 or 的短路求值特性。

3.2 编写程序,运行后用户输入 4 位整数作为年份,判断其是否为闰年。如果年份能被 400 整除,则为闰年;如果年份能被 4 整除但不能被 100 整除也为闰年。

3.3 Python 提供了两种基本的循环结构:________和________。

3.4 编写程序,生成一个包含 50 个随机整数的列表,然后删除其中所有奇数。(提示:从后向前删)

3.5 编写程序,生成一个包含 20 个随机整数的列表,然后对其中偶数下标的元素进行降序排列,奇数下标的元素不变。(提示:使用切片)

3.6 编写程序，用户从键盘输入小于 1000 的整数，对其进行因式分解。例如，10＝2×5，60＝2×2×3×5。

3.7 编写程序，至少使用两种不同的方法计算 100 以内所有奇数的和。

3.8 编写程序，输出所有由 1、2、3、4 这四个数字组成的素数，并且在每个素数中每个数字只使用一次。

3.9 编写程序，实现分段函数计算，如下表所示。

| x | y |
|---|---|
| x＜0 | 0 |
| 0＜＝x＜5 | x |
| 5＜＝x＜10 | 3x－5 |
| 10＜＝x＜20 | 0.5x－2 |
| 20＜＝x | 0 |

# 第 4 章　字符串与正则表达式

最早的字符串编码是美国标准信息交换码 ASCII，仅对 10 个数字、26 个大写英文字母、26 个小写英文字母及一些其他符号进行了编码。ASCII 采用 1 个字节来对字符进行编码，因此最多只能表示 256 个符号。

随着信息技术的发展和信息交换的需要，各国的文字都需要进行编码，于是分别设计了不同的编码格式，并且编码格式之间有着较大的区别，其中常见的编码有 UTF-8、GB2312、GBK、CP936 等。采用不同的编码格式意味着不同的表示和存储形式，把同一字符存入文件时，写入的内容可能会不同，在理解其内容时必须了解编码规则并进行正确的解码。其中，UTF-8 编码是国际通用的编码，以 1 个字节表示英语字符（兼容 ASCII），以 3 个字节表示中文及其他语言，UTF-8 对全世界所有国家需要用到的字符进行了编码。

GB2312 是我国制定的中文编码标准，使用 1 个字节表示英语，2 个字节表示中文；GBK 是 GB2312 的扩充，而 CP936 是微软在 GBK 基础上开发的编码方式。GB2312、GBK 和 CP936 都是使用 2 个字节表示中文，UTF-8 使用 3 个字节表示中文。在众多编码方案中，Unicode 是不同编码格式之间进行互相转换的基础。

在 Windows 平台上使用 Python 2.x 时，input() 函数从键盘输入的字符串默认为 GBK 编码，而 Python 程序中的字符串编码则使用 # coding 显式地指定，常用的方式有：

```
#coding=utf-8
#coding:GBK
#-*-coding:utf-8-*-
```

Python 2.x 对中文支持不够，因此常常需要在不同的编码之间互相转换，例如下面是 Python 2.7.8 环境执行的结果：

```
>>> s1='中国'
>>> s1
'\xd6\xd0\xb9\xfa'
>>> len(s1)
4
>>> s2=s1.decode('GBK')
>>> s2
u'\u4e2d\u56fd'
>>> len(s2)
2
>>> s3=s2.encode('UTF-8')
>>> s3
'\xe4\xb8\xad\xe5\x9b\xbd'
>>> len(s3)
```

```
6
>>> print s1, s2, s3
中国 中国 中国
```

Python 3.x 中则完全支持中文，无论是一个数字、英文字母，还是一个汉字，都按一个字符对待和处理。例如在 Python 3.4.2 环境中执行下面的代码，从代码中可以看到，在 Python 3.x 中甚至可以使用中文作为变量名。

```
>>> s='中国山东烟台'
>>> len(s)
6
>>> s='SDIBT'
>>> len(s)
5
>>> s='中国山东烟台 SDIBT'
>>> len(s)
11
>>> 姓名='张三'
>>> 年龄=40
>>> print(姓名)
张三
>>> print(年龄)
40
```

## 4.1 字符串

在 Python 中，字符串属于不可变序列类型，使用单引号、双引号、三单引号或三双引号作为界定符，并且不同的界定符之间可以互相嵌套。除了支持序列通用方法（包括比较、计算长度、元素访问、分片等操作）以外，字符串类型还支持一些特有的操作方法，例如，格式化操作、字符串查找、字符串替换等等。但由于字符串属于不可变序列，不能对字符串对象进行元素增加、修改与删除等操作。字符串对象提供的 replace() 和 translate() 方法并不是对原字符串直接进行修改替换，而是返回一个修改替换后的结果字符串，并不对原字符串做任何改动。

Python 支持字符串驻留机制，即：对于短字符串，将其赋值给多个不同的对象时，内存中只有一个副本，多个对象共享该副本。这一点不适用于长字符串，即长字符串不遵守驻留机制，下面的代码演示了短字符串和长字符串在这方面的区别。

```
>>> a='1234'
>>> b='1234'
>>> id(a)==id(b)
True
>>> a='1234' * 50
>>> b='1234' * 50
>>> id(a)==id(b)
False
```

如果需要判断一个变量 s 是否为字符串，应使用 isinstance(s, basestring)。在 Python 2.x 中，字符串有 str 和 unicode 两种，其基类都是 basestring，在 Python 3.x 中合二为一了。在 Python 3.x 中，程序源文件默认为 UTF-8 编码，全面支持中文，字符串对象不再有 encode 和 decode 方法。甚至，在 Python 3.x 中可以使用中文作为变量名。下面的代码演示了 Python 2.7.8 中的字符串类型：

```
>>> import types
>>> types.StringType
<type 'str'>
>>> basestring
<type 'basestring'>
>>> s='hello world'
>>> isinstance(s, basestring)
True
>>> type(s)
<type 'str'>
>>> type(s)==types.StringType
True
>>> ss=u'hello world'
>>> type(ss)
<type 'unicode'>
>>> isinstance(ss, basestring)
True
>>> type(ss)==types.UnicodeType
True
>>> type(ss)==types.StringType
False
```

### 4.1.1 字符串格式化

如果需要将其他类型数据转换为字符串或另一种数字格式，或者嵌入其他字符串或模板中再进行输出，就需要用到字符串格式化。Python 中字符串格式化的格式如图 4-1 所示，“%”符号之前的部分为格式字符串，之后的部分为需要进行格式化的内容。

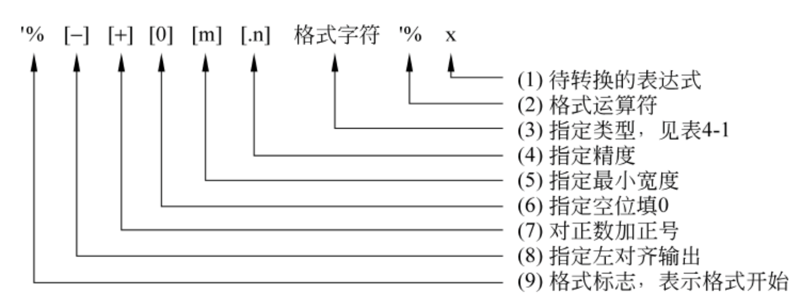



图 4-1 字符串格式化

与其他语言一样，Python 支持大量的格式字符，常见的格式字符如表 4-1 所示。

**表 4-1 格式字符**

| 格式字符 | 说　明 | 格式字符 | 说　明 |
|---|---|---|---|
| %s | 字符串（采用 str()的显示） | %x | 十六进制整数 |
| %r | 字符串（采用 repr()的显示） | %e | 指数（基底写为 e） |
| %c | 单个字符 | %E | 指数（基底写为 E） |
| %b | 二进制整数 | %f、%F | 浮点数 |
| %d | 十进制整数 | %g | 指数(e)或浮点数（根据显示长度） |
| %i | 十进制整数 | %G | 指数(E)或浮点数（根据显示长度） |
| %o | 八进制整数 | %% | 字符"%" |

下面的代码简单演示了字符串格式化的用法：

```
>>> x=1235
>>> so="%o" %x
>>> so
"2323"
>>> sh="%x" %x
>>> sh
"4d3"
>>> se="%e" %x
>>> se
"1.235000e+03"
>>> chr(ord("3")+1)
"4"
>>> "%s"%65                  #类似于 str()
"65"
>>> "%s"%65333
"65333"
>>> '%d,%c'%(65, 65)         #使用元组对字符串进行格式化,按位置进行对应
'65,A'
>>> "%d"%"555"               #试图将字符串转换为整数进行输出,抛出异常
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in<module>
    "%d"%"555"
TypeError: %d format: a number is required, not str
>>> int('555')               #可以使用 int()函数将合法的数字字符串转换为整数
555
>>> '%s'%[1, 2, 3]
'[1, 2, 3]'
>>> str((1, 2, 3))           #可以使用 str()函数将任意类型数据转换为字符串
```

```
'(1, 2, 3)'
>>> str([1, 2, 3])
'[1, 2, 3]'
```

除了上面介绍的字符串格式化方法，目前 Python 社区更推荐使用 format()方法进行格式化，该方法更加灵活，不仅可以使用位置进行格式化，还支持使用与位置无关的参数名字来进行格式化，并且支持序列解包格式化字符串，为程序员提供了非常大的方便。例如：

```
>>> print("The number {0:,} in hex is: {0:#x}, the number {1} in oct is {1:#o}".
format(5555, 55))
The number 5,555 in hex is: 0x15b3, the number 55 in oct is 0o67
>>> print("The number {1:,} in hex is: {1:#x}, the number {0} in oct is {0:#o}".
format(5555, 55))
The number 55 in hex is: 0x37, the number 5555 in oct is 0o12663
>>> print("my name is {name}, my age is {age}, and my QQ is {qq}".format(name=
"Dong Fuguo", qq="306467355", age=37))
my name is Dong Fuguo, my age is 37, and my QQ is 306467355
>>> position=(5, 8, 13)
>>> print("X:{0[0]};Y:{0[1]};Z:{0[2]}".format(position))
X:5;Y:8;Z:13
>>> weather=[("Monday", "rain"), ("Tuesday", "sunny"), ("Wednesday",
"sunny"), ("Thursday", "rain"), ("Friday", "Cloudy")]
>>> formatter="Weather of '{0[0]}' is '{0[1]}'".format
>>> for item in map(formatter, weather):
    print(item)
Weather of 'Monday' is 'rain'
Weather of 'Tuesday' is 'sunny'
Weather of 'Wednesday' is 'sunny'
Weather of 'Thursday' is 'rain'
Weather of 'Friday' is 'Cloudy'
```

关于内置函数 map()的介绍可以参考 5.7 节的介绍。另外，上面最后一段代码也可以改为下面的写法：

```
>>> for item in weather:
    print(formatter(item))
Weather of 'Monday' is 'rain'
Weather of 'Tuesday' is 'sunny'
Weather of 'Wednesday' is 'sunny'
Weather of 'Thursday' is 'rain'
Weather of 'Friday' is 'Cloudy'
```

### 4.1.2 字符串常用方法

字符串是非常重要的数据类型，Python 提供了大量的函数支持字符串操作。本节通

过大量示例来演示部分函数的用法，可以使用 dir("")查看所有字符串操作函数列表，并使用内置函数 help()查看每个函数的帮助。因为字符串也是 Python 序列的一种，除了本节介绍的字符串处理函数，很多 Python 内置函数也支持对字符串的操作，例如用来计算序列长度的 len()方法，用来比较序列大小的 cmp()方法，等等。

**1. find()、rfind()、index()、rindex()、count()**

find()和 rfind()方法分别用来查找一个字符串在另一个字符串指定范围(默认是整个字符串)中首次和最后一次出现的位置，如果不存在则返回 −1；index()和 rindex()方法用来返回一个字符串在另一个字符串指定范围中首次和最后一次出现的位置，如果不存在则抛出异常；count()方法用来返回一个字符串在另一个字符串中出现的次数。

```
>>> s="apple,peach,banana,peach,pear"
>>> s.find("peach")              #返回第一次出现的位置
6
>>> s.find("peach", 7)           #从指定位置开始查找
19
>>> s.find("peach", 7, 20)       #在指定范围中进行查找
-1
>>> s.rfind('p')                 #从字符串尾部向前查找
25
>>> s.index('p')                 #返回首次出现位置
1
>>> s.index('pe')
6
>>> s.index('pear')
25
>>> s.index('ppp')               #指定子字符串不存在时抛出异常
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in<module>
    s.index('ppp')
ValueError: substring not found
>>> s.count('p')                 #统计子字符串出现次数
5
>>> s.count('pp')
1
>>> s.count('ppp')
0
```

**2. split()、rsplit()、partition()、rpartition()**

split()和 rsplit()方法分别用来以指定字符为分隔符，从字符串左端和右端开始将其分割成多个字符串，并返回包含分割结果的列表；partition()和 rpartition()用来以指定字符串为分隔符将原字符串分割为 3 部分，即分隔符前的字符串、分隔符字符串、分隔符后

的字符串，如果指定的分隔符不在原字符串中，则返回原字符串和两个空字符串。

```
>>> s="apple,peach,banana,pear"
>>> li=s.split(",")                    #使用逗号进行分割
>>> li
["apple", "peach", "banana", "pear"]
>>> s.partition(',')
('apple', ',', 'peach,banana,pear')
>>> s.rpartition(',')
('apple,peach,banana', ',', 'pear')
>>> s.rpartition('banana')
('apple,peach,', 'banana', ',pear')
>>> s="2014-10-31"
>>> t=s.split("-")
>>> t
['2014', '10', '31']
>>> map(int, t)
[2014, 10, 31]
```

对于 split()和 rsplit()方法，如果不指定分隔符，则字符串中的任何空白符号(包括空格、换行符、制表符等等)都将被认为是分隔符，返回包含最终分割结果的列表。

```
>>> s='hello world \n\n My name is Dong   '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Dong']
>>> s='\n\nhello world \n\n\n My name is Dong   '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Dong']
>>> s='\n\nhello\t\t world \n\n\n My name\t is Dong   '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Dong']
```

split()和 rsplit()方法还允许指定最大分割次数，例如：

```
>>> s='\n\nhello\t\t world \n\n\n My name is Dong   '
>>> s.split(None, 1)
['hello', 'world \n\n\n My name is Dong   ']
>>> s.rsplit(None, 1)
['\n\nhello\t\t world \n\n\n My name is', 'Dong']
>>> s.split(None, 2)
['hello', 'world', 'My name is Dong   ']
>>> s.rsplit(None, 2)
['\n\nhello\t\t world \n\n\n My name', 'is', 'Dong']
>>> s.split(None, 5)
['hello', 'world', 'My', 'name', 'is', 'Dong   ']
>>> s.split(None, 6)
```

```
['hello', 'world', 'My', 'name', 'is', 'Dong']
```

**3. join()**

与 split()相反，join()方法用来将列表中多个字符串进行连接，并在相邻两个字符串之间插入指定字符。

```
>>> li=["apple", "peach", "banana", "pear"]
>>> sep=","
>>> s=sep.join(li)
>>> s
"apple,peach,banana,pear"
```

使用运算符“+”也可以连接字符串，但效率较低，应优先使用 join()方法。下面的 Python 2.7.8 代码演示了二者之间速度的差异。

```
import timeit

strlist=['This is a long string that will not keep in memory.' for n in xrange
(100)]

def use_join():
    return ''.join(strlist)

def use_plus():
    result=''
    for strtemp in strlist:
        result=result+strtemp
    return result

if __name__=='__main__':
    times=1000
    jointimer=timeit.Timer('use_join()', 'from __main__ import use_join')
    print 'time for join:', jointimer.timeit(number=times)
    plustimer=timeit.Timer('use_plus()', 'from __main__ import use_plus')
    print 'time for plus:', plustimer.timeit(number=times)
```

该代码分别使用 join()函数和“+”对 100 个字符串进行连接，并重复运行 1000 次，然后输出每种方法所使用的时间，运行结果为：

```
time for join: 0.00395874865103
time for plus: 0.0260573301694
```

上面代码使用 timeit 模块的 Timer 类对代码运行时间进行测试。另外，该模块还支持下面代码演示的用法。

```
>>> import timeit
```

```
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.6054277848162267
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.5314926897133567
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.33093395948368
```

#### 4. lower()、upper()、capitalize()、title()、swapcase()

这几个方法分别用来将字符串转换为小写、大写字符串、将字符串首字母变为大写、将每个单词的首字母变为大写以及大小写互换。

```
>>> s="What is Your Name?"
>>> s2=s.lower()
>>> s2
"what is your name?"
>>> s.upper()
"WHAT IS YOUR NAME?"
>>> s2.capitalize()
"What is your, name?"
>>> s.title()
'What Is Your Name?'
>>> s.swapcase()
'wHAT IS yOUR nAME?'
```

#### 5. replace()

该方法用来替换字符串中指定字符或子字符串的所有重复出现，每次只能替换一个字符或一个子字符串。

```
>>> s="中国,中国"
>>> print(s)
中国,中国
>>> s2=s.replace("中国", "中华人民共和国")
>>> print(s2)
中华人民共和国,中华人民共和国
```

#### 6. maketrans()、translate()

maketrans()方法用来生成字符映射表，而 translate()方法则按映射表关系转换字符串并替换其中的字符，使用这两个方法的组合可以同时处理多个不同的字符，replace()方法则无法满足这一要求。下面的代码演示了这两个方法的用法，当然还可以定义自己的字符映射表，然后用来对字符串进行加密。

```
>>> import string
#将字符"abcdef123"一一对应地转换为"uvwxyz@#$"
```

```
#在 Python 3.x 中应写作 table=''.maketrans("abcdef123", "uvwxyz@#$")
>>> table=string.maketrans("abcdef123", "uvwxyz@#$")
>>> s="Python is a greate programming language. I like it!"
>>> s.translate(table)
"Python is u gryuty progrumming lunguugy. I liky it!"
>>> s.translate(table, "gtm")          #第二个参数表示要删除的字符
"Pyhon is u ryuy proruin lunuuy. I liky i!"
```

### 7. strip()、rstrip()、lstrip()

这几个方法分别用来删除两端、右端或左端的空白字符或连续的指定字符。

```
>>> s=" abc  "
>>> s2=s.strip()                       #删除空白字符
>>> s2
"abc"
>>> '\n\nhello world   \n\n'.strip()   #删除空白字符
'hello world'
>>> "aaaassddf".strip("a")             #删除指定字符
"ssddf"
>>> "aaaassddf".strip("af")
"ssdd"
>>> "aaaassddfaaa".rstrip("a")         #删除字符串右端指定字符
'aaaassddf'
>>> "aaaassddfaaa".lstrip("a")         #删除字符串左端指定字符
'ssddfaaa'
```

### 8. eval()

内置函数 eval()尝试把任意字符串转化为 Python 表达式并进行求值。

```
>>> eval("3+4")
7
>>> a=3
>>> b=5
>>> eval('a+b')
8
>>> import math
>>> eval('help(math.sqrt)')
Help on built-in function sqrt in module math:
sqrt(...)
    sqrt(x)
Return the square root of x.
>>> eval('math.sqrt(3)')
1.7320508075688772
```

```
>>> eval('aa')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in<module>
    eval('aa')
  File "<string>", line 1, in<module>
NameError: name 'aa' is not defined
```

使用 eval()时要注意的一个问题是,它可以计算任意合法表达式的值,如果用户巧妙地构造输入的字符串,可以执行任意外部程序,例如,下面的代码运行后可以启动记事本程序:

```
>>> a=input("Please input a value:")
Please input a value:"__import__('os').startfile(r'C:\Windows\\notepad.
exe')"
>>> eval(a)
```

是不是非常危险啊? 如果你觉得这没什么,再执行下面的代码试试,然后看看当前工作目录中多了什么,当然还可以调用命令来删除这个文件夹或其他文件,或者精心构造其他字符串来达到特殊目的。

```
>>> eval("__import__('os').system('md testtest')")
```

### 9. 关键字 in

与列表、元组、字典、集合一样,也可以使用关键字 in 和 not in 来判断一个字符串是否出现在另一个字符串中,返回 True 或 False。

```
>>> "a" in "abcde"
True
>>> 'ab' in 'abcde'
True
>>> "j" in "abcde"
False
```

### 10. startswith()、endswith()

这两个方法用来判断字符串是否以指定字符串开始或结束。在 2.1.9 节中介绍列表推导式时用到过,请自行查阅。这两个方法可以接收两个整数参数来限定字符串的检测范围,例如

```
>>> s='Beautiful is better than ugly.'
>>> s.startswith('Be')
True
>>> s.startswith('Be', 5)
False
>>> s.startswith('Be', 0, 5)
```

```
True
```

另外，这两个方法还可以接收一个字符串元组作为参数来表示前缀或后缀，例如下面的代码可以列出指定文件夹下所有扩展名为.bmp、.jpg 或.gif 的图片。

```
>>> import os
>>> [filename for filename in os.listdir(r'D:\\') if filename.endswith
(('.bmp', '.jpg', '.gif'))]
```

**11. isalnum()、isalpha()、isdigit()、isspace()、isupper()、islower()**

用来测试字符串是否为数字或字母、是否为字母、是否为数字字符、是否为空白字符、是否为大写字母以及是否为小写字母。

```
>>> '1234abcd'.isalnum()
True
>>> '1234abcd'.isalpha()
False
>>> '1234abcd'.isdigit()
False
>>> 'abcd'.isalpha()
True
>>> '1234.0'.isdigit()
False
>>> '1234'.isdigit()
True
```

**12. center()、ljust()、rjust()**

返回指定宽度的新字符串，原字符串居中、左对齐或右对齐出现在新字符串中，如果指定的宽度大于字符串长度，则使用指定的字符（默认为空格）进行填充。

```
>>> 'Hello world!'.center(20)
'    Hello world!    '
>>> 'Hello world!'.center(20, '=')
'====Hello world!===='
>>> 'Hello world!'.ljust(20, '=')
'Hello world!========'
>>> 'Hello world!'.rjust(20, '=')
'========Hello world!'
```

### 4.1.3 字符串常量

在 string 模块中定义了多个字符串常量，包括数字字符、标点符号、英文字母、大写字

母、小写字母等等，用户可以直接使用这些常量。下面的代码在 Python 2.7.8 中运行：

```
>>> import string
>>> string.digits              #数字字符常量
'0123456789'
>>> string.punctuation         #标点符号常量
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.letters             #英文字母常量
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
>>> string.printable           #可打印字符
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()
*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
>>> string.lowercase           #小写字母
'abcdefghijklmnopqrstuvwxyz'
>>> string.uppercase           #大写字母
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

在 2.3.4 节曾经演示过字符串常量的用法，这里再给出了一个示例，下面的 Python 3.4.2 代码演示了 8 位长度随机密码生成算法的原理。

```
>>> import string
>>> x=string.digits+string.ascii_letters+string.punctuation
>>> x
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()
*+,-./:;<=>?@[\\]^_`{|}~'
>>> import random
>>> ''.join([random.choice(x) for i in range(8)])
'H\\{.#=)g'
>>> ''.join([random.choice(x) for i in range(8)])
'(CrZ[44M'
>>> ''.join([random.choice(x) for i in range(8)])
'o_?[M>iF'
>>> ''.join([random.choice(x) for i in range(8)])
'n<[I)5V@'
```

### 4.1.4 可变字符串

在 Python 中，字符串属于不可变对象，不支持原地修改，如果需要修改其中的值，只能重新创建一个新的字符串对象。然而，如果确实需要一个支持原地修改的 unicode 数据对象，可以使用 io.StringIO 对象或 array 模块。

```
>>> import io
>>> s="Hello, world"
>>> sio=io.StringIO(s)
```

```
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a=array.array('u', s)
>>> print(a)
array('u', 'Hello, world')
>>> a[0]='y'
>>> print(a)
array('u', 'yello, world')
>>> a.tounicode()
'yello, world'
```

## 4.2 正则表达式

正则表达式是字符串处理的有力工具和技术,正则表达式使用预定义的特定模式去匹配一类具有共同特征的字符串,主要用于字符串处理,可以快速、准确地完成复杂的查找、替换等处理要求。

Python 中,re 模块提供了正则表达式操作所需要的功能。本节首先介绍正则表达式的基础知识,然后介绍 re 模块提供的正则表达式函数与对象的用法。

### 4.2.1 正则表达式元字符

正则表达式由元字符及其不同组合来构成,通过巧妙地构造正则表达式可以匹配任意字符串,并完成复杂的字符串处理任务。常用的正则表达式元字符如表 4-2 所示。

**表 4-2 正则表达式常用元字符**

| 元字符 | 功能说明 |
| --- | --- |
| . | 匹配除换行符以外的任意单个字符 |
| * | 匹配位于 * 之前的 0 个或多个字符 |
| + | 匹配位于+之前的一个或多个字符 |
| \| | 匹配位于\|之前或之后的字符 |
| ^ | 匹配行首,匹配以^后面的字符开头的字符串 |

续表

| 元字符 | 功 能 说 明 |
| --- | --- |
| $ | 匹配行尾，匹配以 $ 之前的字符结束的字符串 |
| ? | 匹配位于? 之前的 0 个或 1 个字符 |
| \ | 表示位于\之后的为转义字符 |
| [] | 匹配位于[]中的任意一个字符 |
| - | 用在[]之内用来表示范围 |
| () | 将位于()内的内容作为一个整体来对待 |
| {} | 按{}中的次数进行匹配 |
| \b | 匹配单词头或单词尾 |
| \B | 与\b 含义相反 |
| \d | 匹配任何数字，相当于[0-9] |
| \D | 与\d 含义相反 |
| \s | 匹配任何空白字符 |
| \S | 与\s 含义相反 |
| \w | 匹配任何字母、数字以及下划线，相当于[a-zA-Z0-9_] |
| \W | 与\w 含义相反 |

如果以“\”开头的元字符与转义字符相同，则需要使用“\\”，或者使用原始字符串，即在字符串前加上字符“r”或“R”。原始字符串可以减少用户的输入，主要用于正则表达式和文件路径字符串的情况，但如果字符串以一个斜线“\”结束，则需要多写一个斜线，即以“\\”结束。

具体应用时，可以单独使用某种类型的元字符，但处理复杂字符串时，经常需要将多个正则表达式元字符进行组合，下面给出了几个简单的示例：

- 最简单的正则表达式是普通字符串，可以匹配自身。
- '[pjc]ython'可以匹配'python'、'jython'、'cython'。
- '[a-zA-Z0-9]'可以匹配一个任意大小写字母或数字。
- '[^abc]'可以一个匹配任意除'a'、'b'、'c'之外的字符。
- 'python|perl'或'p(ython|erl)'都可以匹配'python'或'perl'。
- 子模式后面加上问号表示可选。r'(http://)?(www\.)?python\.org'只能匹配'http://www.python.org'、'http://python.org'、'www.python.org'和'python.org'。
- '^http'只能匹配所有以'http'开头的字符串。
- (pattern)*：允许模式重复 0 次或多次。
- (pattern)+：允许模式重复 1 次或多次。
- (pattern){m，n}：允许模式重复 m～n 次。

在具体构造正则表达式时，要注意到可能会发生的错误，尤其是涉及到特殊字符的时候。例如下面这段代码(完整代码参见 4.2.6 节的例 4-2)，作用是用来匹配 Python 程序中的运算符，但是因为有些运算符与正则表达式的元字符相同而引起歧义，如果处理不当则会造成理解错误，需要进行必要的转义处理。

```
>>> import re
>>> symbols=[',', '+', '-', '*', '/', '//', '**', '>>', '<<', '+=', '-=',
'*=', '/=']
>>> for i in symbols:
    patter=re.compile(r'\s*'+i+r'\s*')
Traceback (most recent call last):
  File "<pyshell#11>", line 2, in<module>
    patter=re.compile(r'\s*'+i+r'\s*')
  File "C:\python27\lib\re.py", line 190, in compile
    return _compile(pattern, flags)
  File "C:\python27\lib\re.py", line 244, in _compile
    raise error, v #invalid expression
error: multiple repeat
>>> for i in symbols:
    patter=re.compile(r'\s*'+re.escape(i)+r'\s*')
正常执行
```

### 4.2.2 re 模块主要方法

在 Python 中，主要使用 re 模块来实现正则表达式的操作。该模块的常用方法如表 4-3 所示，具体使用时，既可以直接使用 re 模块的方法进行字符串处理，也可以将模式编译为正则表达式对象，然后使用正则表达式对象的方法来操作字符串。

**表 4-3 re 模块常用方法**

| 方　　法 | 功 能 说 明 |
|---|---|
| compile(pattern[,flags]) | 创建模式对象 |
| search(pattern,string[,flags]) | 在整个字符串中寻找模式，返回 match 对象或 None |
| match(pattern,string[,flags]) | 从字符串的开始处匹配模式，返回 match 对象或 None |
| findall(pattern,string[,flags]) | 列出字符串中模式的所有匹配项 |
| split(pattern,string[,maxsplit=0]) | 根据模式匹配项分割字符串 |
| sub(pat,repl,string[,count=0]) | 将字符串中所有 pat 的匹配项用 repl 替换 |
| escape(string) | 将字符串中所有特殊正则表达式字符转义 |

其中函数参数 flags 的值可以是 re.I(忽略大小写)、re.L、re.M(多行匹配模式)、re.S(使元字符“.”匹配任意字符，包括换行符)、re.U(匹配 Unicode 字符)、re.X(忽略模

式中的空格，并可以使用#注释)的不同组合(使用“|”进行组合)。

### 4.2.3 直接使用 re 模块方法

可以直接使用 re 模块的方法来实现正则表达式操作，本节通过几个具体的示例来简单演示其用法。

```
>>> import re
>>> text='alpha. beta....gamma delta'
>>> re.split('[\. ]+', text)
['alpha', 'beta', 'gamma', 'delta']
>>> re.split('[\. ]+', text, maxsplit=2)          #分割2次
['alpha', 'beta', 'gamma delta']
>>> re.split('[\. ]+', text, maxsplit=1)          #分割1次
['alpha', 'beta....gamma delta']
>>> pat='[a-zA-Z]+'
>>> re.findall(pat, text)                         #查找所有单词
['alpha', 'beta', 'gamma', 'delta']
>>> pat='{name}'
>>> text='Dear {name}...'
>>> re.sub(pat, 'Mr.Dong', text)                  #字符串替换
'Dear Mr.Dong...'
>>> s='a s d'
>>> re.sub('a|s|d', 'good', s)                    #字符串替换
'good good good'
>>> re.escape('http://www.python.org')            #字符串转义
'http\\:\\/\\/www\\.python\\.org'
>>> print(re.match('done|quit', 'done'))          #匹配成功
<_sre.SRE_Match object at 0x00B121A8>
>>> print(re.match('done|quit', 'done!'))         #匹配成功
<_sre.SRE_Match object at 0x00B121A8>
>>> print(re.match('done|quit', 'doe!'))          #匹配不成功
None
>>> print(re.match('done|quit', 'd!one!'))        #匹配不成功
None
>>> print(re.match('done|quit', 'd!one!done'))    #匹配不成功
None
>>> print(re.search('done|quit', 'd!one!done'))   #匹配成功
<_sre.SRE_Match object at 0x0000000002D03D98>
```

下面的代码使用不同的方法删除字符串中多余的空格，如果遇到连续多个空格则只保留一个。

```
>>> import re
```

```
>>> s='aaa      bb      c d e   fff     '
>>> re.sub('\s+', ' ', s)                           #直接使用 re 模块的字符串替换方法
'aaa bb c d e fff '
>>> re.split('[\s]+', s)
['aaa', 'bb', 'c', 'd', 'e', 'fff', '']
>>> re.split('[\s]+', s.strip())                    #同时删除了字符串尾部的空格
['aaa', 'bb', 'c', 'd', 'e', 'fff']
>>> ' '.join(re.split('[\s]+', s.strip()))
'aaa bb c d e fff'
>>> ' '.join(re.split('\s+', s.strip()))
'aaa bb c d e fff'
>>> re.sub('\s+', ' ', s.strip())
'aaa bb c d e fff'
>>> s.split()                                       #也可以不使用正则表达式
['aaa', 'bb', 'c', 'd', 'e', 'fff']
>>> ' '.join(s.split())
'aaa bb c d e fff'
```

下面的代码使用以“\”开头的元字符来实现字符串的特定搜索。

```
>>> import re
>>> example='ShanDong Institute of Business and Technology is a very beautiful
school.'
>>> re.findall('\\ba.+? \\b', example)      #以 a 开头的完整单词
['and', 'a ']
>>> re.findall('\\ba\w*\\b', example)
['and', 'a']
>>> re.findall('\\Bo.+? \\b', example)      #含有字母 o 的单词中第一个非首字母 o 后
                                             面的剩余部分
['ong', 'ology', 'ool']
>>> re.findall('\\b\w.+? \\b', example)     #所有单词
['ShanDong', 'Institute', 'of', 'Business', 'and', 'Technology', 'is', 'a ',
'very', 'beautiful', 'school']
>>> re.findall(r'\b\w.+? \b', example)      #使用原始字符串,减少需要输入的符号数量
['ShanDong', 'Institute', 'of', 'Business', 'and', 'Technology', 'is', 'a ',
'very', 'beautiful', 'school']
>>> re.split('\s', example)                 #使用任何空白字符分割字符串
['ShanDong', 'Institute', 'of', 'Business', 'and', 'Technology', 'is', 'a',
'very', 'beautiful', 'school.']
>>> re.findall('\d\.\d\.\d', 'Python 2.7.8')        #查找并返回 x.x.x 形式的数字
['2.7.8']
>>> re.findall('\d\.\d\.\d', 'Python 2.7.8,Python 3.4.2')
['2.7.8', '3.4.2']
```

### 4.2.4 使用正则表达式对象

首先使用 re 模块的 compile()方法将正则表达式编译生成正则表达式对象,然后再

使用正则表达式对象提供的方法进行字符串处理，使用编译后的正则表达式对象可以提高字符串处理速度。

正则表达式对象的 match(string[, pos[, endpos]])方法用于在字符串开头或指定位置进行搜索，模式必须出现在字符串开头或指定位置；search(string[, pos[, endpos]])方法用于在整个字符串或指定范围中进行搜索；findall(string[, pos[, endpos]])方法用于在字符串中查找所有符合正则表达式的字符串并以列表形式返回。

```
>>> import re
>>> example='ShanDong Institute of Business and Technology'
>>> pattern=re.compile(r'\bB\w+\b')          #以 B 开头的单词
>>> pattern.findall(example)
['Business']
>>> pattern=re.compile(r'\w+g\b')            #以 g 结尾的单词
>>> pattern.findall(example)
['ShanDong']
>>> pattern=re.compile(r'\b[a-zA-Z]{3}\b')#查找 3 个字母长的单词
>>> pattern.findall(example)
['and']
>>> pattern.match(example)                   #从字符串开头开始匹配，不成功，没有返回值
>>> pattern.search(example)                  #在整个字符串中搜索，成功
<_sre.SRE_Match object at 0x01228EC8>
>>> pattern=re.compile(r'\b\w*a\w*\b') #查找所有含有字母 a 的单词
>>> pattern.findall(example)
['ShanDong', 'and']
>>> text="He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)               #查找所有副词
['carefully', 'quickly']
```

正则表达式对象的 sub(repl,string[,count=0])和 subn(repl,string[,count=0])方法用来实现字符串替换功能。

```
>>> example='''Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.'''
>>> pattern=re.compile(r'\bb\w*\b', re.I)
>>> print pattern.sub('*', example)           #将以字母“b”和“B”开头的单词替换为“*”
* is * than ugly.
Explicit is * than implicit.
Simple is * than complex.
```

```
Complex is * than complicated.
Flat is * than nested.
Sparse is * than dense.
Readability counts.
>>> print pattern.sub('*', example, 1)    #只替换一次
 * is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
>>> pattern=re.compile(r'\bb\w*\b')
>>> print pattern.sub('*', example, 1)     #将第一个以字母"b"开头的单词替换为"*"
Beautiful is * than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
```

正则表达式对象的 split(string[, maxsplit=0])方法用来实现字符串分割。

```
>>> example=r'one,two,three.four/five\six? seven[eight]nine|ten'
>>> pattern=re.compile(r'[,./\\?[\]\|]') #指定多个可能的分隔符
>>> pattern.split(example)
['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
>>> example=r'one1two2three3four4five5six6seven7eight8nine9ten'
>>> pattern=re.compile(r'\d+')           #使用数字作为分隔符
>>> pattern.split(example)
['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
>>> example=r'one two    three  four,five.six.seven,eight,nine9ten'
>>> pattern=re.compile(r'[\s,.\d]+')      #允许分隔符重复
>>> pattern.split(example)
['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

### 4.2.5 子模式与 match 对象

使用圆括号"()"表示一个子模式,圆括号内的内容作为一个整体出现,例如"(red)+"可以匹配"redred"、"redredred"等多个重复"red"的情况。

```
>>> telNumber='''Suppose my Phone No. is 0535-1234567,
yours is 010-12345678, his is 025-87654321.'''
```

```
>>> pattern=re.compile(r'(\d{3, 4})-(\d{7, 8})')
>>> pattern.findall(telNumber)
[('0535', '1234567'), ('010', '12345678'), ('025', '87654321')]
```

正则表达式模块或正则表达式对象的 match()方法和 search()方法匹配成功后都会返回 match 对象。match 对象的主要方法有 group()(返回匹配的一个或多个子模式内容)、groups()(返回一个包含匹配的所有子模式内容的元组)、groupdict()(返回包含匹配的所有命名子模式内容的字典)、start()(返回指定子模式内容的起始位置)、end()(返回指定子模式内容的结束位置的前一个位置)、span()(返回一个包含指定子模式内容起始位置和结束位置前一个位置的元组)等等。例如,下面的代码使用 re 模块的 search()方法返回的 match 对象来删除字符串中指定内容。

```
>>> email="tony@ tiremove_thisger.net"
>>> m=re.search("remove_this", email)
>>> email[:m.start()]+email[m.end():]
'tony@ tiger.net'
```

下面的代码演示了 match 对象的 group()方法的用法。

```
>>> m=re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)          #返回整个模式内容
'Isaac Newton'
>>> m.group(1)          #返回第 1 个子模式内容
'Isaac'
>>> m.group(2)          #返回第 2 子模式内容
'Newton'
>>> m.group(1, 2)       #返回指定的多个子模式内容
('Isaac', 'Newton')
>>> m=re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

下面的代码演示了 match 对象的 groups()方法的用法。

```
>>> m=re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

下面的代码演示了 match 对象的 groupdict()方法。

```
>>> m=re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

下面的代码使用正则表达式的 search()返回的 match 对象提取字符串中的电话号码。

```
import re

telNumber='''Suppose my Phone No. is 0535-1234567, yours is 010-12345678, his
is 025-87654321.'''
pattern=re.compile(r'(\d{3, 4})-(\d{7, 8})')
index=0
while True:
    matchResult=pattern.search(telNumber, index)
    if not matchResult:
        break
    print '-'*30
    print 'Success:'
    for i in range(3):
        print 'Searched content:', matchResult.group(i),\
        ' Start from:', matchResult.start(i), 'End at:', matchResult.end(i),\
          ' Its span is:', matchResult.span(i)
    index=matchResult.end(2)
```

上面程序的运行结果为：

```
------------------------------
Success:
Searched content: 0535-1234567  Start from: 24 End at: 36  Its span is: (24, 36)
Searched content: 0535  Start from: 24 End at: 28  Its span is: (24, 28)
Searched content: 1234567  Start from: 29 End at: 36  Its span is: (29, 36)
------------------------------
Success:
Searched content: 010-12345678  Start from: 47 End at: 59  Its span is: (47, 59)
Searched content: 010  Start from: 47 End at: 50  Its span is: (47, 50)
Searched content: 12345678  Start from: 51 End at: 59  Its span is: (51, 59)
------------------------------
Success:
Searched content: 025-87654321  Start from: 68 End at: 80  Its span is: (68, 80)
Searched content: 025  Start from: 68 End at: 71  Its span is: (68, 71)
Searched content: 87654321  Start from: 72 End at: 80  Its span is: (72, 80)
```

使用子模式扩展语法可以实现更加复杂的字符串处理，常用的扩展语法如表 4-4 所示。

**表 4-4 子模式扩展语法**

| 语　　法 | 功 能 说 明 |
| --- | --- |
| (?P<groupname>) | 为子模式命名 |
| (?iLmsux) | 设置匹配标志，可以是几个字母的组合，每个字母含义与编译标志相同 |
| (?:...) | 匹配但不捕获该匹配的子表达式 |

续表

| 语　　法 | 功 能 说 明 |
| --- | --- |
| (?P=groupname) | 表示在此之前的命名为 groupname 的子模式 |
| (?#...) | 表示注释 |
| (?=...) | 用于正则表达式之后,表示如果=后的内容在字符串中出现则匹配,但不返回=之后的内容 |
| (?!...) | 用于正则表达式之后,表示如果!后的内容在字符串中不出现则匹配,但不返回!之后的内容 |
| (?<=...) | 用于正则表达式之前,与(?=...)含义相同 |
| (?<!...) | 用于正则表达式之前,与(?!...)含义相同 |

下面通过几个示例来演示子模式扩展语法的应用。

```
>>> import re
>>> exampleString='''There should be one--and preferably only one --obvious
way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.'''
>>> pattern=re.compile(r'(?<=\w\s)never(?=\s\w)') #查找不在句子开头和结尾的单词
>>> matchResult=pattern.search(exampleString)
>>> matchResult.span()
(172, 177)
>>> pattern=re.compile(r'(?<=\w\s)never')         #查找位于句子末尾的单词
>>> matchResult=pattern.search(exampleString)
>>> matchResult.span()
(156, 161)
>>> pattern=re.compile(r'(?:is\s)better(\sthan)')
                                        #查找前面是 is 的 better than 组合
>>> matchResult=pattern.search(exampleString)
>>> matchResult.span()
(141, 155)
>>> matchResult.group(0)                         #组 0 表示整个模式
'is better than'
>>> matchResult.group(1)
' than'
>>> pattern=re.compile(r'\b(?i)n\w+\b')          #查找以 n 或 N 字母开头的所有单词
>>> index=0
>>> while True:
    matchResult=pattern.search(exampleString, index)
    if not matchResult:
        break
```

```
    print matchResult.group(0), ':', matchResult.span(0)
    index=matchResult.end(0)
not : (92, 95)
Now : (137, 140)
never : (156, 161)
never : (172, 177)
now : (205, 208)
>>> pattern=re.compile(r'(?<!not\s)be\b')        #查找前面没有单词 not 的单词 be
>>> index=0
>>> while True:
    matchResult=pattern.search(exampleString, index)
    if not matchResult:
        break
    print matchResult.group(0), ':', matchResult.span(0)
    index=matchResult.end(0)
be : (13, 15)
>>> exampleString[13:20]                          #验证一下结果是否正确
'be one-'
>>> pattern=re.compile(r'(\b\w*(?P<f>\w+)(?P=f)\w*\b)')
                                                  #查找具有连续相同字母的单词
>>> index=0
>>> while True:
    matchResult=pattern.search(exampleString, index)
    if not matchResult:
        break
    print matchResult.group(0), ':', matchResult.group(2)
    index=matchResult.end(0)+1
unless : s
better : t
better : t
>>> s
'aabc abcd abbcd abccd abcdd'
>>> p=re.compile(r'(\b\w*(?P<f>\w+)(?P=f)\w*\b)')
>>> p.findall(s)
[('aabc', 'a'), ('abbcd', 'b'), ('abccd', 'c'), ('abcdd', 'd')]
```

### 4.2.6 正则表达式应用案例精选

在本章的最后，我们通过两个在实际开发中非常有用的案例来演示字符串处理以及正则表达式的使用。

**例 4-1** 标识符提取。将本案例代码存为文件 FindIdentifiersFromPyFile.py，运行时按照提示输入要检测的 Python 源程序文件，该程序会自动提取出 Python 源文件中所

有的类名、函数名以及各种变量名。当然,你可以很容易地修改本程序以实现所需要的更为复杂的业务逻辑。

```
import re
import os

classes={}
functions=[]
variables={'normal':{}, 'parameter':{}, 'infor':{}}

def _identifyClassNames(index, line):
    '''parameter index is the line number of line,
      parameter line is a line of code of the file to check'''
    pattern=re.compile(r'(?<=class\s)\w+(?=.*?:)')
    matchResult=pattern.search(line)
    if not matchResult:
        return
    className=matchResult.group(0)
    classes[className]=classes.get(className, [])
    classes[className].append(index)

def _identifyFunctionNames(index, line):
    pattern=re.compile(r'(?<=def\s)(\w+)\((.*?)\)(?=:)')
    matchResult=pattern.search(line)
    if not matchResult:
        return
    functionName=matchResult.group(1)
    functions.append((functionName, index))
    parameters=matchResult.group(2).split(r', ')
    if parameters[0]=='':
        return
    for v in parameters:
        variables['parameter'][v]=variables['parameter'].get(v, [])
        variables['parameter'][v].append(index)

def _identifyVariableNames(index, line):
    #find normal variables, including the case: a, b=3, 5
    pattern=re.compile(r'\b(.*?)(?=\s=)')
    matchResult=pattern.search(line)
    if matchResult:
        vs=matchResult.group(1).split(r', ')
        for v in vs:
            #consider the case 'if variable==value'
            if 'if ' in v:
```

```
                v=v.split()[1]
            #consider the case: 'a[3]=3'
            if '[' in v:
                v=v[0:v.index('[')]
            variables['normal'][v]=variables['normal'].get(v, [])
            variables['normal'][v].append(index)

    #find the variables in for statements
    pattern=re.compile(r'(?<=for\s)(.*?)(?=\sin)')
    matchResult=pattern.search(line)
    if matchResult:
        vs=matchResult.group(1).split(r', ')
        for v in vs:
            variables['infor'][v]=variables['infor'].get(v, [])
            variables['infor'][v].append(index)

def output():
    print '='*30
    print 'The class names and their line numbers are:'
    for key, value in classes.items():
        print key, ':', value

    print '='*30
    print 'The function names and their line numbers are:'
    for i in functions:
        print i[0], ':', i[1]

    print '='*30
    print 'The normal variable names and their line numbers are:'
    for key, value in variables['normal'].items():
        print key, ':', value
    print '-'*20
    print 'The parameter names and their line numbers in functions are:'
    for key, value in variables['parameter'].items():
        print key, ':', value
    print '-'*20
    print 'The variable names and their line numbers in for statements are:'
    for key, value in variables['infor'].items():
        print key, ':', value

#suppose the lines of comments less than 50
def comments(index):
    for i in range(50):
        line=allLines[index+i].strip()
```

```
        if line.endswith('''''''') or line.endswith(''''''''):
            return i+1

if __name__=='__main__':
     fileName=input('Please input the file name (including the full path if
neccesary:')
    if not os.path.isfile(fileName):
        print 'Your input is not a file.'
    if not fileName.endswith('.py'):
        print 'Sorry. I can only check Python source file.'
    allLines=[]
    with open(fileName, 'r') as fp:
        allLines=fp.readlines()

    index=0
    totalLen=len(allLines)
    while index<totalLen:
        line=allLines[index]
        #strip the blank characters at both end of line
        line=line.strip()
        #ignore the comments starting with '#'
        if line.startswith('#'):
            index+=1
            continue
        #ignore the comments between ''' or """
        if line.startswith('''''''') or line.startswith(''''''''):
            index+=comments(index)
            continue
        #identify identifiers
        _identifyClassNames(index+1, line)
        _identifyFunctionNames(index+1, line)
        _identifyVariableNames(index+1, line)
        index+=1
    output()
```

**例 4-2** Python 程序规范性检查。下面的代码可以用来检查 Python 程序的规范性(详见 1.5 节),其中用到了正则表达式、文件对象、字符串处理方法以及列表切片等知识。

```
#-*-coding:utf-8-*-
#Filename: CheckCodeFormats.py
import sys
import re

def checkFormats(lines, desFileName):
```

```
fp=open(desFileName, 'w')
for i, line in enumerate(lines):
    print '=' * 30
    print 'Line:', i+1
    if line.strip().startswith('#'):
        print ' ' * 10+'Pass.'
        fp.write(line)
        continue

    flag=True

    #check operator symbols
    symbols=[',', '+', '-', '*', '/', '//', '**', '>>', '<<', '+=', '-=',
    '*=', '/=']
    temp_line=line
    for symbol in symbols:
        pattern=re.compile(r'\s*'+re.escape(symbol)+r'\s*')
        temp_line=pattern.split(temp_line)
        sep=' '+symbol+' '
        temp_line=sep.join(temp_line)
    if line !=temp_line:
        flag=False
        print ' ' * 10+'You may miss some blank spaces in this line.'
        line=temp_line

    #check import statement
    if line.strip().startswith('import'):
        if ',' in line:
            flag=False
            print ' ' * 10+"You'd better import one module at a time."
            temp_line=line.strip()
            modules=temp_line[temp_line.index(' ')+1:]
            modules=modules.strip()
            pattern=re.compile(r'\s*,\s*')
            modules=pattern.split(modules)
            temp_line=''
            for module in modules:
                temp_line+=line[:line.index('import')]+'import '+module
                +'\n'
            line=temp_line

        pri_line=lines[i-1].strip()
        if pri_line and (not pri_line.startswith('import')) and \
           (not pri_line.startswith('#')):
            flag=False
            print ' ' * 10+'You should add a blank line before this line.'
```

```
                line='\n'+line

            after_line=lines[i+1].strip()
            if after_line and (not after_line.startswith('import')):
                flag=False
                print ' ' * 10+'You should add a blank line after this line.'
                line=line+'\n'

        #check if there is a blank line before new funtional code block,
        including the class/function definition
        if line.strip() and not line.startswith(' ') and i>0:
            pri_line=lines[i-1]
            if pri_line.strip() and pri_line.startswith(' '):
                flag=False
                print ' ' * 10+"You'd better add a blank line before this line."
                line='\n'+line

        if flag:
            print ' ' * 10+'Pass.'
        fp.write(line)
    fp.close()

if __name__=='__main__':
    fileName='CheckCodeFormats.py'#sys.argv[1]
    fileLines=[]
    with open(fileName, 'r') as fp:
        fileLines=fp.readlines()
    desFileName=fileName[:-3]+'_new.py'
    checkFormats(fileLines, desFileName)

    #check the ratio of comment lines to all lines
    comments=[line for line in fileLines if line.strip().startswith('#')]
    ratio=len(comments) * 1.0/len(fileLines)
    if ratio<=0.3:
        print '=' * 30
        print 'Comments in the file is less than 30%.'
        print 'Perhaps you should add some comments at appropriate position.'
```

## 本章小结

（1）在 Python 中，字符串属于不可变序列类型，使用单引号、双引号、三单引号或三双引号作为界定符，并且不同的界定符之间可以互相嵌套。

（2）字符串属于有序不可变序列，不支持任何方法来直接修改字符串的内容。

（3）在格式化字符串时，优先考虑使用 format()方法。

(4) Python 3.x 全面支持中文,Python 2.x 对中文支持还不够,在处理中文时需要在不同的编码格式之间进行必要的转换。

(5) 对于短字符串,Python 支持驻留机制,即相同的字符串在内存中只有一个副本,长字符串不具有这个特性。

(6) 虽然字符串属于不可变序列,但支持使用 replace()方法、maketrans()和 translate()方法以及正则表达式的方法进行内容替换操作,这些方法都返回新字符串,并不对原字符串做任何修改。

(7) 字符串的 split()和 rsplit()方法分别用来以指定字符为分隔符,从字符串左端和右端开始将其分割成多个字符串,并返回包含分割结果的列表;join()方法用来将列表中多个字符串进行连接,并在相邻两个字符串之间插入指定字符。

(8) 对用户输入的字符串进行 eval()操作时可能会有安全漏洞,应对用户输入的内容进行必要的检查和过滤。

(9) 在 string 模块中定义了多个字符串常量,包括数字字符、表达符号、英文字母、大写字母、小写字母等等。

(10) 正则表达式是字符串处理的有力工具和技术,可以快速实现字符串的复杂处理。

(11) 可以直接使用 re 模块的方法来进行字符串处理,也可以将模式编译为正则表达式对象,然后使用正则表达式对象的方法来处理字符串。

(12) 正则表达式中的子模式是作为一个整体来对待的,使用子模式扩展语法可以实现更加复杂的字符串处理要求。

(13) 正则表达式对象的 match(string[, pos[, endpos]])方法用于在字符串开头或指定位置进行搜索,模式必须出现在字符串开头或指定位置;search(string[, pos[, endpos]])方法用于在整个字符串或指定范围中进行搜索;匹配成功的话,这两个方法都返回 match 对象,match 对象的主要方法有 group()、groups()、groupdict()、start()、end()、span()等等。

(14) 正则表达式对象的 findall(string[, pos[, endpos]])方法用于在字符串中查找所有符合正则表达式的字符串并以列表形式返回。

## 习题

4.1 假设有一段英文,其中有单独的字母“I”误写为“i”,请编写程序进行纠正。

4.2 假设有一段英文,其中有单词中间的字母“i”误写为“I”,请编写程序进行纠正。

4.3 有一段英文文本,其中有单词连续重复了 2 次,编写程序检查重复的单词并只保留一个。例如,文本内容为“This is is a desk.”,程序输出为“This is a desk.”。

4.4 简单解释 Python 的字符串驻留机制。

4.5 编写程序,用户输入一段英文,然后输出这段英文中所有长度为 3 个字母的单词。

# 第 5 章　函数设计与使用

在实际开发中,有很多操作是完全相同或者是非常相似的,仅仅是要处理的数据不同而已,因此经常会在不同的代码位置多次执行相似或完全相同的代码块。从软件设计和代码复用的角度来讲,很显然,直接将该代码块复制到多个相应的位置然后进行简单修改绝对不是一个好主意。虽然这样使得多份复制的代码可以彼此独立地进行修改,但这样不仅增加了代码量,使得程序文件变大,也增加了代码理解和代码维护的难度,更重要的是为代码测试和纠错带来了很大的困难。一旦被复制的代码块在将来某天被发现存在问题而需要修改,则必须对所有的复制都做同样正确的修改,这在实际中是很难完成的一项任务。由于代码量的大幅度增加,导致代码之间的关系更加复杂,很可能在修补旧漏洞的同时又引入了新漏洞。因此,应尽量减少使用直接复制代码块的方式来实现复用。

解决上述问题的一种常用方式是设计和编写函数,另一种是面向对象程序设计中的类,本章介绍函数的设计与使用,第 6 章介绍面向对象程序设计。将可能需要反复执行的代码封装为函数,并在需要执行该段代码功能的地方进行调用,不仅可以实现代码的复用,更重要的是可以保证代码的一致性,只需要修改该函数代码则所有调用位置均得到体现。当然,在实际开发中,需要对函数进行良好的设计和优化才能充分发挥其优势。在编写函数时,有很多原则需要参考和遵守,例如,不要在同一个函数中执行太多的功能,尽量只让其完成一个高度相关且大小合适的功能,以提高模块的内聚性。另外,尽量减少不同函数之间的隐式耦合,例如减少全局变量的使用,使得函数之间仅通过调用和参数传递来显式体现其相互关系。

在编写函数时,函数体中代码的编写与前面章节介绍的基本一致,只是对代码进行了封装并增加了函数调用、传递参数、返回计算结果等外围接口,这也正是本章讲解的重点。另外,由于 Python 程序是解释执行的,因此如果函数或代码编写的有问题,只有在被调用和执行时才可能被发现,甚至包括某些语法错误。另外,还有可能传递某些类型的参数时执行正确,而传递另一些类型的参数时则可能会出现错误。出现这样的情况有多种可能的原因,例如,不同的参数值可能会使得函数执行不同的路径,或者不同的参数类型所支持的操作和运算符不同,等等。所以,在进行代码测试时一定要注意,一次或几次运行正常并不表示代码编写的没有问题,必须要进行尽可能完全的测试,尽量满足各种覆盖性要求,尽量在代码发布之前发现和解决更多的潜在问题。

## 5.1　函数定义与调用

在 Python 中,定义函数的语法如下:

```
def 函数名([参数列表]):
```

```
    '''注释'''
    函数体
```

在 Python 中使用 def 关键字来定义函数，然后是一个空格和函数名称，接下来是一对圆括号，在圆括号内是形式参数列表，如果有多个参数则使用逗号分隔开，圆括号之后是一个冒号和换行，最后是必要的注释和函数体代码。定义函数时需要注意的问题是：

(1) 函数形参不需要声明其类型，也不需要指定函数返回值类型；

(2) 即使该函数不需要接收任何参数，也必须保留一对空的圆括号；

(3) 括号后面的冒号必不可少；

(4) 函数体相对于 def 关键字必须保持一定的空格缩进。

最后，Python 允许嵌套定义函数，并且所有包含__call__()方法的类的对象均被认为是可调用的，这部分内容请参见 5.8 节的讨论。

例如，下面的函数用来计算斐波那契数列中小于参数 n 的所有值：

```
def fib(n):
    a, b=1, 1
    while a<n:
        print(a, end=' ')
        a, b=b, a+b
    print()
```

该函数的调用方式为：

```
fib(1000)
```

在定义函数时，开头部分的注释并不是必需的，但是如果为函数的定义加上一段注释的话，可以为用户提供友好的提示和使用帮助。例如，把上面生成斐波那契数列的函数定义修改为下面的形式，在函数开头加上一段注释。

```
>>> def fib(n):
    '''accept an integer n.
       return the numbers less than n in Fibonacci sequence.'''
    a, b=1, 1
    while a<n:
        print(a, end=' ')
        a, b=b, a+b
    print()
```

如此一来，在调用该函数时，输入左侧圆括号之后，立刻就会得到该函数的使用说明，如图 5-1 所示。

```
>>> def fib(n):
	'''accept an integer n.
	   return the numbers less than n in Fibonacci sequence.'''
	a, b = 1, 1
	while a < n:
	    print(a, end=' ')
	    a, b = b, a+b
	print()

>>> fib(
        (n)
        accept an integer n.
        return the numbers less than n in Fibonacci sequence.
```

图 5-1 使用注释来为用户提示函数使用说明

## 5.2 形参与实参

函数定义时圆括号内是使用逗号分隔开的形参列表(parameters),一个函数可以没有形参,但是定义时一对圆括号必须要有,表示这是一个函数并且不接收参数。函数调用时向其传递实参(arguments),根据不同的参数类型,将实参的值或引用传递给形参。

例如,在5.1节中定义函数fib()时括号内的"n"就是该函数的形参,而调用该函数时括号内的"1000"则是传递给该函数的实参。

在定义函数时,对参数个数并没有限制,如果有多个形参,则需要使用逗号进行分隔。例如下面的函数用来接收两个参数,并输出其中的最大值。

```
def printMax(a, b):
    if a>b:
        pirnt(a, 'is the max')
    else:
        print(b, 'is the max')
```

当然,这里只是为了演示,而忽略了一些细节,如果输入的参数不支持比较运算,则会出错,可以参考后面第8章中介绍的异常处理结构来解决这个问题。

对于绝大多数情况下,在函数内部直接修改形参的值不会影响实参。例如下面的示例:

```
>>> def addOne(a):
    print(a)
    a+=1
    print(a)
>>> a=3
>>> addOne(a)
3
4
>>> a
3
```

从运行结果可以看出，在函数内部修改了形参 a 的值，但是当函数运行结束以后，实参 a 的值并没有被修改，可以参考 5.5 节中关于变量作用域的讨论。当然，在有些情况下，可以通过特殊的方式在函数内部修改实参的值，例如下面的代码：

```
>>> def modify(v):              #修改列表元素值
    v[0]=v[0]+1
>>> a=[2]
>>> modify(a)
>>> a
[3]
>>> def modify(v, item):        #为列表增加元素
    v.append(item)
>>> a=[2]
>>> modify(a, 3)
>>> a
[2, 3]
>>> def modify(d):              #修改字典元素值或为字典增加元素
    d['age']=38
>>> a={'name':'Dong', 'age':37, 'sex':'Male'}
>>> a
{'age': 37, 'name': 'Dong', 'sex': 'Male'}
>>> modify(a)
>>> a
{'age': 38, 'name': 'Dong', 'sex': 'Male'}
```

也就是说，如果传递给函数的是 Python 可变序列，并且在函数内部使用下标或其他方式为可变序列增加、删除元素或修改元素值时，修改后的结果是可以反映到函数之外的，即实参也得到了相应的修改。

## 5.3 参数类型

在 Python 中，函数参数有很多种，主要可以分为普通参数、默认值参数、关键参数、可变长度参数等等。Python 函数的定义也非常灵活，在定义函数时不需要指定参数的类型，形参的类型完全由调用者传递的实参类型以及 Python 解释器的理解和推断来决定，类似于某些语言中的泛型；同样，也不需要指定函数的返回值类型，这将由函数中的 return 语句来决定。函数的返回值类型由 return 语句返回值的类型来决定，如果函数中没有 return 语句或者没有执行到 return 语句而返回或者执行了不带任何值的 return 语句，则函数都默认为返回空值 None。

### 5.3.1 默认值参数

在定义函数时，Python 支持默认值参数，即在定义函数时为形参设置默认值。在调

用带有默认值参数的函数时，可以不用为设置了默认值的形参进行传值，此时函数将会直接使用函数定义时设置的默认值。默认值参数与 5.3.3 节介绍的可变长度参数可以实现类似于函数重载的目的。带有默认值参数的函数定义语法如下：

```
def 函数名(…,形参名=默认值):
    函数体
```

调用带有默认值参数的函数时，可以不对默认值参数进行赋值，也可以通过显式赋值来替换其默认值，具有较大的灵活性。如果需要的话，可以使用“函数名.func_defaults”（在 Python 3.x 中使用“函数名.__defaults__”）随时查看函数所有默认值参数的当前值，其返回值为一个元组，其中的元素依次表示每个默认值参数的当前值。例如下面的函数定义：

```
>>> def say( message, times=1 ):
    print((message+' ') * times)
>>> say.func_defaults
(1,)
```

调用该函数时，如果只为第一个参数传递实参，则第二个参数使用默认值“1”，如果为第二个参数传递实参，则不再使用默认值“1”，而是使用调用者显式传递的值。

```
>>> say('hello')
hello
>>> say('hello', 3)
hello hello hello
>>> say('hi', 7)
hi hi hi hi hi hi hi
```

再例如，下面的函数使用指定分隔符将列表中所有字符串元素连接成一个字符串，如果调用者没有指定分隔符，则默认使用空格。

```
>>> def Join(List, sep=None):
    return (sep or ' ').join(List)
>>> aList=['a', 'b', 'c']
>>> Join(aList)
'a b c'
>>> Join(aList, ',')
'a,b,c'
```

需要注意的是，在定义带有默认值参数的函数时，默认值参数必须出现在函数形参列表的最右端，且任何一个默认值参数右边都不能再出现非默认值参数。例如下面的示例，前两个函数不符合这一要求，从而导致函数定义失败，如图 5-2 所示。

另外，特别需要注意的是，多次调用函数并且不为默认值参数传递值时，默认值参数只在第一次调用时进行解释。对于列表、字典这样复杂类型的默认值参数，这一点可能会导致很严重的逻辑错误，而这种错误或许会耗费较多的精力来定位和纠正。例如下面的

代码：

```
def demo(newitem, old_list=[]):
    old_list.append(newitem)
return old_list

print(demo('5', [1, 2, 3, 4]))
print(demo('aaa', ['a', 'b']))
print(demo('a'))
print(demo('b'))
```

可以试运行一下上面的代码，仔细看看结果，是否能发现问题呢？然后再把代码修改为下面的样子，再试运行一下，看看区别在哪里。然后再仔细阅读本节前面的内容，应该会发现答案。

```
def demo(newitem, old_list=None):
    if old_list is None:
        old_list=[]
    old_list.append(newitem)
return old_list

print(demo('5', [1, 2, 3, 4]))
print(demo('aaa', ['a', 'b']))
print(demo('a'))
print(demo('b'))
```

```
>>> def f(a=3,b,c=5):
        print a,b,c

SyntaxError: non-default argument follows default argument
>>> def f(a=3,b):
        print a,b

SyntaxError: non-default argument follows default argument
>>> def f(a,b,c=5):
        print a,b,c


>>>
```

图 5-2 带有默认值参数的函数定义

### 5.3.2 关键参数

关键参数主要指调用函数时的参数传递方式，而与函数定义无关。

通过关键参数可以按参数名字传递值，实参顺序可以和形参顺序不一致，但不影响参数值的传递结果，避免了用户需要牢记参数位置和顺序的麻烦，使得函数的调用和参数传递更加灵活方便。

```
>>> def demo(a, b, c=5):
```

```
    print(a, b, c)
>>> demo(3, 7)
3 7 5
>>> demo(a=7, b=3, c=6)
7 3 6
>>> demo(c=8, a=9, b=0)
9 0 8
```

### 5.3.3 可变长度参数

可变长度参数在定义函数时主要有两种形式：*parameter 和**parameter，前者用来接收任意多个实参并将其放在一个元组中，后者接收类似于关键参数一样显式赋值形式的多个实参并将其放入字典中。

下面的代码演示了第一种形式可变长度参数的用法，即无论调用该函数时传递了多少实参，一律将其放入元组中：

```
>>> def demo(*p):
    print(p)
>>> demo(1, 2, 3)
(1, 2, 3)
>>> demo(1, 2, 3, 4, 5, 6, 7)
(1, 2, 3, 4, 5, 6, 7)
```

下面的代码则演示了第二种形式可变长度参数的用法，即在调用该函数时自动将接收的参数转换为字典：

```
>>> def demo(**p):
    for item in p.items():
        print(item)
>>> demo(x=1, y=2, z=3)
('y', 2)
('x', 1)
('z', 3)
```

下面的代码演示了定义函数时几种不同形式的参数混合使用的用法。需要注意的是，虽然 Python 完全支持你这样做，但是除非真的很必要，否则请不要这样用，因为这会使得代码非常混乱而严重降低可读性，并导致程序查错非常困难。另外，一般而言，一个函数如果可以接收很多参数，很可能是函数设计得不好，例如，函数功能过多，需要进行必要的拆分和重新设计，以满足高内聚的要求。

```
>>> def func_4(a, b, c=4, *aa, **bb):
    print((a, b, c))
    print(aa)
    print(bb)
```

```
>>> func_4(1, 2, 3, 4, 5, 6, 7, 8, 9, xx='1', yy='2', zz=3)
(1, 2, 3)
(4, 5, 6, 7, 8, 9)
{'yy': '2', 'xx': '1', 'zz': 3}
>>> func_4(1, 2, 3, 4, 5, 6, 7, xx='1', yy='2', zz=3)
(1, 2, 3)
(4, 5, 6, 7)
{'yy': '2', 'xx': '1', 'zz': 3}
```

### 5.3.4 参数传递时的序列解包

为含有多个变量的函数传递参数时，可以使用 Python 列表、元组、集合、字典以及其他可迭代对象作为实参，并在实参名称前加一个星号，Python 解释器将自动进行解包，然后传递给多个单变量形参。但需要注意的是，如果使用字典对象作为实参，则默认使用字典的“键”，如果需要将字典中“键-值对”作为参数则需要使用 items()方法，如果需要将字典的“值”作为参数则需要调用字典的 values()方法。最后，请务必保证实参中元素个数与形参个数相等，否则将出现错误。

```
>>> def demo(a, b, c):
    print(a+b+c)
>>> seq=[1, 2, 3]
>>> demo(*seq)
6
>>> tup=(1, 2, 3)
>>> demo(*tup)
6
>>> dic={1:'a', 2:'b', 3:'c'}
>>> demo(*dic)
6
>>> Set={1, 2, 3}
>>> demo(*Set)
6
>>> demo(*dic.values())
abc
```

## 5.4 return 语句

return 语句用来从一个函数中返回并结束函数的执行，同时还可以通过 return 语句从函数中返回一个任意类型的值。不论 return 语句出现在函数的什么位置，一旦得到执行将直接结束函数的执行。如果函数没有 return 语句或者执行了不返回任何值的 return 语句，Python 将认为该函数以 return None 结束，即返回空值。

```
def maximum( x, y ):
    if x>y:
          return x
    else:
          return y
```

作为使用者,在调用函数时,一定要注意函数有没有返回值,以及是否会对参数的值进行修改。例如第 2 章介绍过的列表对象方法 sort()属于原地操作,没有返回值,而内置函数 sorted()则返回排序后的列表,并不对原列表做任何修改。

```
>>> a_list=[1, 2, 3, 4, 9, 5, 7]
>>> print(sorted(a_list))
[1, 2, 3, 4, 5, 7, 9]
>>> print(a_list)
[1, 2, 3, 4, 9, 5, 7]
>>> print(a_list.sort())
None
>>> print(a_list)
[1, 2, 3, 4, 5, 7, 9]
```

## 5.5 变量作用域

变量起作用的代码范围称为变量的作用域,不同作用域内同名变量之间互不影响。一个变量在函数外部定义和在函数内部定义,其作用域是不同的,函数内部定义的变量一般为局部变量,而不属于任何函数的变量一般为全局变量。一般而言,局部变量的引用比全局变量速度快,应优先考虑使用,前面章节介绍过类似问题,此处不再赘述。另外,除非真的有必要,否则应尽量避免使用全局变量,因为全局变量会增加不同函数之间的隐式耦合度,从而降低代码可读性,并使得代码测试和纠错变得很困难。

在函数内定义的普通变量只在该函数内起作用,称为局部变量。当函数运行结束后,在该函数内部定义的局部变量被自动删除而不可访问。在函数内部定义的全局变量当函数结束以后仍然存在并且可以访问。

如果想要在函数内部修改一个定义在函数外的变量值,那么这个变量就不能是局部的,其作用域必须为全局的,能够同时作用于函数内外,称为全局变量,可以通过 global 来声明或定义。这分两种情况:

- 一个变量已在函数外定义,如果在函数内需要修改这个变量的值,并将这个赋值结果反映到函数之外,可以在函数内用 global 声明这个变量为全局变量,明确声明要使用已定义的同名全局变量。
- 在函数内部直接使用 global 关键字将一个变量声明为全局变量,即使在函数外没有定义该全局变量,在调用这个函数之后,将自动增加新的全局变量。

或者说,也可以这么理解:在函数内如果只引用某个变量的值而没有为其赋新值,该变量为(隐式的)全局变量;如果在函数内任意位置有为变量赋新值的操作,该变量即被认

为是(隐式的)局部变量,除非在函数内显式地用关键字 global 进行声明。

我们通过下面的示例代码来演示局部变量和全局变量的用法。

```
>>> def demo():
    global x              #声明或创建全局变量
    x=3                   #修改全局变量的值
    y=4                   #局部变量
    print(x, y)
>>> x=5                   #在函数外部定义了全局变量 x
>>> demo()                #本次调用修改了全局变量 x 的值
3  4
>>> x
3
>>> y                     #局部变量在函数运行结束之后自动删除
出错信息
NameError: name 'y' is not defined
>>> del x                 #删除了全局变量 x
>>> x
出错信息
NameError: name 'x' is not defined
>>> demo()                #本次调用创建了全局变量
3  4
>>> x
3
>>> y                     #局部变量在函数调用和执行结束后自动删除,在函数外部不可访问
出错信息
NameError: name 'y' is not defined
```

如果局部变量与全局变量具有相同的名字,那么该局部变量会在自己的作用域内隐藏同名的全局变量,例如下面的代码所演示。

```
>>> def demo():
    x=3                   #创建了局部变量,并自动隐藏了同名的全局变量
>>> x=5
>>> x
5
>>> demo()
>>> x
5
```

最后,如果需要在同一个程序的不同模块之间共享全局变量的话,可以编写一个专门的模块来实现这一目的。例如,假设在模块 A.py 中有如下变量定义:

```
global_variable=0
```

而在模块 B.py 中包含以下语句:

```
import A
A.global_variable=1
```

在模块C.py中有以下语句：

```
import A
print(A.global_variable)
```

从而实现了在不同模块之间共享全局变量的目的。

## 5.6 lambda表达式

lambda表达式可以用来声明匿名函数，即没有函数名字的临时使用的小函数。lambda表达式只可以包含一个表达式，不允许包含其他复杂的语句，但在表达式中可以调用其他函数，并支持默认值参数和关键参数，该表达式的计算结果就是函数的返回值。下面的代码演示了不同情况下lambda表达式的应用。

```
>>> f=lambda x, y, z: x+y+z
>>> print(f(1, 2, 3))
6
>>> g=lambda x, y=2, z=3: x+y+z        #含有默认值参数
>>> print(g(1))
6
>>> print(g(2, z=4, y=5))              #调用时使用关键参数
11
>>> L=[(lambda x: x**2), (lambda x: x**3), (lambda x: x**4)]
>>> print(L[0](2), L[1](2), L[2](2))
4 8 16
>>> D={'f1':(lambda: 2+3), 'f2':(lambda: 2*3), 'f3':(lambda: 2**3)}
>>> print(D['f1'](), D['f2'](), D['f3']())
5 6 8
>>> L=[1, 2, 3, 4, 5]
>>> print(map((lambda x: x+10), L))   #没有名字的lambda表达式
[11, 12, 13, 14, 15]
>>> L
[1, 2, 3, 4, 5]
>>> def demo(n):
    return n*n
>>> demo(5)
25
>>> a_list=[1, 2, 3, 4, 5]
>>> map(lambda x: demo(x), a_list)    #包含函数调用并且没有名字的lambda表达式
[1, 4, 9, 16, 25]
>>> data=list(range(20))
>>> print(data)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
>>> import random
>>> random.shuffle(data)
>>> data
[4, 3, 11, 13, 12, 15, 9, 2, 10, 6, 19, 18, 14, 8, 0, 7, 5, 17, 1, 16]
>>> data.sort(key=lambda x: x)        #用在列表的 sort()方法中
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> data.sort(key=lambda x: len(str(x)))
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> data.sort(key=lambda x: len(str(x)), reverse=True)
>>> data
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

在使用 lambda 表达式时，要注意变量作用域带来的问题，例如，在下面的代码中，变量 x 是在外部作用域中定义的，对 lambda 表达式而言不是局部变量，从而导致出现了错误。

```
>>> r=[]
>>> for x in range(10):
    r.append(lambda: x**2)
>>> r[0]()
81
>>> r[1]()
81
>>> r[2]()
81
```

若修改为下面的代码，则可以得到正确的结果。

```
>>> r=[]
>>> for x in range(10):
    r.append(lambda n=x: n**2)
>>> r[0]()
0
>>> r[1]()
1
>>> r[5]()
25
>>> r[8]()
64
```

## 5.7 案例精选

**例 5-1** 编写函数计算圆的面积。

```
from math import pi as PI
```

```
import types
def CircleArea(r):
    if isinstance(r, int) or isinstance(r, float): #确保接收的参数为数值
        return PI * r * r
    else:
        print('You must give me an integer or float as radius.')

print(CircleArea(3))
```

**例 5-2** 编写函数，接收任意多个实数，返回一个元组，其中第一个元素为所有参数的平均值，其他元素为所有参数中大于平均值的实数。

```
def demo( * para):
    avg=sum(para)/len(para) #注意 Python 2.x 与 Python 3.x 对除法运算符“/”的解释不同
    g=[i for i in para if i>avg]
    return (avg,)+tuple(g)

print(demo(1, 2, 3, 4))
```

**例 5-3** 编写函数，接收字符串参数，返回一个元组，其中第一个元素为大写字母个数，第二个元素为小写字母个数。

```
def demo(s):
    result=[0, 0]
    for ch in s:
        if 'a'<=ch<='z':
            result[1]+=1
        elif 'A'<=ch<='Z':
            result[0]+=1
    return result

print(demo('aaaabbbbC'))
```

**例 5-4** 编写函数，接收包含 20 个整数的列表 lst 和一个整数 k 作为参数，返回新列表。处理规则为：将列表 lst 中下标 k 之前的元素逆序，下标 k 之后的元素逆序，然后将整个列表 lst 中的所有元素逆序。

```
def demo(lst, k):
    x=lst[:k]
    x.reverse()
    y=lst[k:]
    y.reverse()
    r=x+y
    r.reverse()
    return r
```

```
lst=list(range(1, 21))
print(lst)
print(demo(lst, 5))
```

**例 5-5** 编写函数，接收整数参数 t，返回斐波那契数列中大于 t 的第一个数。

```
def demo(t):
    a, b=1, 1
    while b<t:
        a, b=b, a+b
    else:
        return b

print(demo(50))
```

**例 5-6** 编写函数，接收一个包含若干整数的列表参数 lst，返回一个元组，其中第一个元素为列表 lst 中的最小值，其余元素为最小值在列表 lst 中的下标。

```
import random

def demo(lst):
    m=min(lst)
    result=(m,)
    for index, value in enumerate(lst):
        if value==m:
            result=result+(index,)
    return result

x=[random.randint(1, 20) for i in range(50)]
print(x)
print(demo(x))
```

**例 5-7** 编写函数，接收一个整数 t 为参数，打印杨辉三角的前 t 行。

```
def demo(t):
    print([1])
    print([1, 1])
    line=[1, 1]
    for i in range(2, t):
        r=[]
        for j in range(0, len(line)-1):
            r.append(line[j]+line[j+1])
        line=[1]+r+[1]
        print(line)

demo(10)
```

上面的代码运行结果为：

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
[1, 8, 28, 56, 70, 56, 28, 8, 1]
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

**例 5-8** 编写函数，接收一个正偶数为参数，输出两个素数，并且这两个素数之和等于原来的正偶数。如果存在多组符合条件的素数，则全部输出。

```
import math

def IsPrime(n):
    m=int(math.sqrt(n))+1
    for i in range(2, m):
        if n%i==0:
            return False
    return True

def demo(n):
    if isinstance(n, int) and n>0 and n%2==0:
        for i in range(3, int(n/2)+1):
            if i%2==1 and IsPrime(i) and IsPrime(n-i):
                print(i, '+', n-i, '=', n)

demo(60)
```

**例 5-9** 编写函数，接收两个正整数作为参数，返回一个数组，其中第一个元素为最大公约数，第二个元素为最小公倍数。

```
def demo(m, n):
    if m>n:
        m, n=n, m
    p=m*n
    while m!=0:
        r=n%m
        n=m
        m=r
    return (int(p/n), n)
```

```
print(demo(20, 30))
```

**例 5-10** 编写函数，接收一个所有元素值互不相等的整数列表 x 和一个整数 n，要求将值为 n 的元素作为支点，将列表中所有值小于 n 的元素全部放到 n 的前面，所有值大于 n 的元素放到 n 的后面。

```
import random

def demo(x, n):
    if n not in x:
        print(n, ' is not an element of ', x)
        return

    i=x.index(n)                        #获取指定元素在列表中的索引
    x[0], x[i]=x[i], x[0]               #将指定元素与第 0 个元素交换
    key=x[0]

    i=0
    j=len(x)-1
    while i<j:
        while i<j and x[j]>=key:        #从后向前寻找第一个比指定元素小的元素
            j -=1
        x[i]=x[j]

        while i<j and x[i]<=key:        #从前向后寻找第一个比指定元素大的元素
            i+=1
        x[j]=x[i]

    x[i]=key

x=list(range(1, 10))
random.shuffle(x)

print(x)
demo(x, 4)
print(x)
```

## 5.8 高级话题

在本章的最后，让我们来看几个高级话题，包括内置 map()、reduce()、filter()、生成器、Python 字节码、函数嵌套定义以及可调用对象的知识。

(1) 内置函数 map()可以将一个单参数函数依次作用到一个序列或迭代器对象的每个元素上，并返回一个列表作为结果，该列表中的每个元素是原序列中元素经过该函数处

理后的结果，该函数不对原序列或迭代器对象做任何修改。

```
>>> map(str, range(5))
['0', '1', '2', '3', '4']
>>> def add5(v):
    return v+5
>>> map(add5, range(10))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

(2) 内置函数 reduce()可以将一个接收两个参数的函数以累积的方式从左到右依次作用到一个序列或迭代器对象的所有元素上。

```
>>> seq=[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> reduce(lambda x, y: x+y, seq)
45
>>> def add(x, y):
    return x+y
>>> reduce(add, range(10))
45
```

上面的代码运行过程如图 5-3 所示。

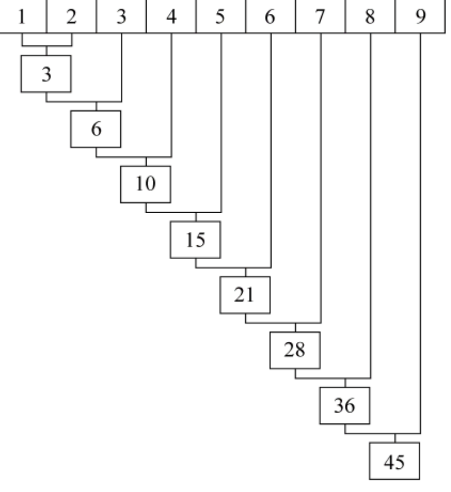



图 5-3　reduce()函数执行过程示意图

类似的运算并不局限于数值类型，例如下面的代码使用前面定义的函数 add()实现了字符串连接。

```
>>> reduce(add, map(str, range(10)))
'0123456789'
```

**注意：**在 Python 3.x 中，使用 reduce()函数需要先从 functools 模块导入，即

```
from functools import reduce
```

(3) 内置函数 filter()将一个单参数函数作用到一个序列上，返回该序列中使得该函数返回值为 True 的那些元素组成的列表、元组或字符串。

```
>>> seq=['foo', 'x41', '?!', '***']
>>> def func(x):
    return x.isalnum()
>>> filter(func, seq)
['foo', 'x41']
>>> seq
['foo', 'x41', '?!', '***']
>>> [x for x in seq if x.isalnum()]
['foo', 'x41']
>>> filter(lambda x: x.isalnum(), seq)
['foo', 'x41']
```

(4) 包含 yield 语句的函数用来创建生成器。迭代器的最大特点是惰性求值，尤其适用于大数据处理。下面的代码演示了如何使用生成器来生成斐波那契数列。

```
>>> def f():
    a, b=1, 1
    while True:
        yield a
        a, b=b, a+b
>>> a=f()
>>> for i in range(10):
    print(a.__next__(), end=' ')
1 1 2 3 5 8 13 21 34 55
```

上面定义的生成器函数还可以这样使用：

```
>>> for i in f():
    if i>100:
        break
    print(i, end=' ')
1 1 2 3 5 8 13 21 34 55 89
```

(5) 使用 dis 模块的功能可以查看函数的字节码指令。

```
>>> def add(n):
    n+=1
    return n
>>> import dis
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (n)
              3 LOAD_CONST               1 (1)
              6 INPLACE_ADD
              7 STORE_FAST               0 (n)

  3          10 LOAD_FAST                0 (n)
             13 RETURN_VALUE
```

(6) 函数嵌套定义与可调用对象。

在 Python 中，函数是可以嵌套定义的。另外，任何包含__call__()方法的类的对象都是可调用的。例如，下面的代码演示了函数嵌套定义的情况：

```
def linear(a, b):
    def result(x):
        return a * x+b
    return result
```

下面的代码演示了可调用对象类的定义：

```
class linear:
```

```
    def __init__(self, a, b):
        self.a, self.b=a, b
    def __call__(self, x):
        return self.a * x+self.b
```

使用上面的两种方式中任何一个，都可以通过以下的方式来定义一个可调用对象：

```
taxes=linear(0.3, 2)
```

然后通过下面的方式来调用该对象：

```
taxes(5)
```

## 本章小结

（1）函数是用来实现代码复用的常用方式。

（2）定义函数时使用关键字 def。

（3）可以在函数定义的开头部分使用一对三单引号增加一段注释来为用户提示函数使用说明。

（4）定义函数时不需要指定其形参类型，而是根据调用函数时传递的实参自动进行推断。

（5）测试函数时，一次或几次运行正确并不能说明函数的设计与实现没有问题，应进行尽可能全面的测试。

（6）对于绝大多数情况，在函数内部直接修改形参的值不会影响实参。

（7）如果传递给函数的是 Python 可变序列，并且在函数内部使用下标或其他方式为可变序列增加、删除元素或修改元素值时，修改后的结果是可以反映到函数之外的，即实参也得到了相应的修改。

（8）定义函数时可以为形参设置默认值，如果调用该函数时不为默认值参数传递参数，将自动使用默认值。

（9）如果使用默认值参数，必须保证默认值参数出现在函数参数列表中的最后，即默认值参数后面不能出现非默认值参数。

（10）多次调用函数并且不为默认值参数传递值时，默认值参数只在第一次调用时进行解释，对于列表、字典这样复杂类型的默认值参数，这一点可能会导致很严重的逻辑错误。

（11）传递参数时可以使用关键参数，避免牢记参数顺序的麻烦。

（12）定义函数时，形参前面加一个星号表示可以接收多个实参并将其放置到一个元组中，形参前面加两个星号表示可以接收多个“键-值对”参数并将其放置到字典中。

（13）为含有多个变量的函数传递参数时，可以使用 Python 列表、元组、集合、字典以及其他可迭代对象作为实参，并在实参名称前加一个星号，Python 解释器将自动进行解包，然后传递给多个单变量形参。

（14）lambda 表达式可以用来创建只包含一个表达式的匿名函数。

(15) 在 lambda 表达式中可以调用其他函数,并支持默认值参数和关键参数。

(16) 定义函数时不需要指定其返回值的类型,而是由 return 语句来决定,如果函数中没有 return 语句或执行了不返回任何值的 return 语句,则 Python 认为该函数返回空值 None。

(17) 在函数内定义的普通变量只在该函数内起作用,称为局部变量。当函数运行结束后,在该函数内部定义的局部变量被自动删除。在函数内部定义的全局变量当函数结束以后仍然存在并且可以访问。

(18) 在函数内部可以通过 global 关键字来声明或者定义全局变量。

## 习题

5.1 运行 5.3.1 节最后的示例代码,查看结果并分析原因。

5.2 编写函数,判断一个整数是否为素数,并编写主程序调用该函数。

5.3 编写函数,接收一个字符串,分别统计大写字母、小写字母、数字、其他字符的个数,并以元组的形式返回结果。

5.4 在函数内部可以通过关键字________来定义全局变量。

5.5 如果函数中没有 return 语句或者 return 语句不带任何返回值,那么该函数的返回值为________。

5.6 调用带有默认值参数的函数时,不能为默认值参数传递任何值,必须使用函数定义时设置的默认值。(判断对错)

5.7 在 Python 程序中,局部变量会隐藏同名的全局变量吗?请编写代码进行验证。

5.8 lambda 表达式只能用来创建匿名函数,不能为这样的函数起名字。(判断对错)

5.9 编写函数,可以接收任意多个整数并输出其中的最大值和所有整数之和。

5.10 编写函数,模拟内置函数 sum()。

5.11 包含________语句的函数可以用来创建生成器。

5.12 编写函数,模拟内置函数 sorted()。

# 第 6 章　面向对象程序设计

面向对象程序设计(Object Oriented Programming,OOP)的思想主要针对大型软件设计而提出,使得软件设计更加灵活,能够很好地支持代码复用和设计复用,并且使得代码具有更好的可读性和可扩展性。面向对象程序设计的一条基本原则是计算机程序由多个能够起到子程序作用的单元或对象组合而成,这大大地降低了软件开发的难度,使得编程就像搭积木一样简单。面向对象程序设计的一个关键性观念是将数据以及对数据的操作封装在一起,组成一个相互依存、不可分割的整体,即对象。对于相同类型的对象进行分类、抽象后,得出共同的特征而形成了类,面向对象程序设计的关键就是如何合理地定义和组织这些类以及类之间的关系。

Python 完全采用了面向对象程序设计的思想,是真正面向对象的高级动态编程语言,完全支持面向对象的基本功能,如封装、继承、多态以及对基类方法的覆盖或重写。但与其他面向对象程序设计语言不同的是,Python 中对象的概念很广泛,Python 中的一切内容都可以称为对象,而不一定必须是某个类的实例。例如,字符串、列表、字典、元组等内置数据类型都具有和类完全相似的语法和用法。创建类时用变量形式表示的对象属性称为数据成员或成员属性,用函数形式表示的对象行为称为成员函数或成员方法,成员属性和成员方法统称为类的成员。

## 6.1　类的定义与使用

### 6.1.1　类定义语法

Python 使用 class 关键字来定义类,class 关键字之后是一个空格,然后是类的名字,再然后是一个冒号,最后换行并定义类的内部实现。类名的首字母一般要大写,当然也可以按照自己的习惯定义类名,但是一般推荐参考惯例来命名,并在整个系统的设计和实现中保持风格一致,这一点对于团队合作尤其重要。例如:

```
class Car:                  #新式类必须有至少一个基类
    def infor(self):
        print(" This is a car ")
```

定义了类之后,可以用来实例化对象,并通过“对象名.成员”的方式来访问其中的数据成员或成员方法,例如下面的代码:

```
>>> car=Car()
>>> car.infor()
 This is a car
```

在 Python 中，可以使用内置方法 isinstance()来测试一个对象是否为某个类的实例，下面的代码演示了 isinstance()的用法。

```
>>> isinstance(car, Car)
True
>>> isinstance(car, str)
False
```

最后，Python 提供了一个关键字 pass，类似于空语句，可以用在类和函数的定义中或者选择结构中。当暂时没有确定如何实现功能，或者为以后的软件升级预留空间，或者其他类型功能时，可以使用该关键字来“占位”。例如下面的代码都是合法：

```
>>> class A:
    pass

>>> def demo():
    pass

>>> if 5>3:
    pass
```

### 6.1.2 self 参数

类的所有实例方法都必须至少有一个名为 self 的参数，并且必须是方法的第一个形参（如果有多个形参的话），self 参数代表将来要创建的对象本身。在类的实例方法中访问实例属性时需要以 self 为前缀，但在外部通过对象名调用对象方法时并不需要传递这个参数，如果在外部通过类名调用对象方法则需要显式为 self 参数传值，参考后面 6.2 节的讨论。

在 Python 中，在类中定义实例方法时将第一个参数定义为 self 只是一个习惯，而实际上类的实例方法中第一个参数的名字是可以变化的，而不必使用 self 这个名字，例如下面的代码：

```
>>> class A:
    def __init__(hahaha, v):
        hahaha.value=v
    def show(hahaha):
        print(hahaha.value)
>>> a=A(3)
>>> a.show()
3
```

### 6.1.3 类成员与实例成员

这里主要指数据成员，或者广义上的属性。可以说属性有两种：一种是实例属性；另

一种是类属性。实例属性一般是指在构造函数__ init __()中定义的,定义和使用时必须以 self 作为前缀;类属性是在类中所有方法之外定义的数据成员。在主程序中(或类的外部),实例属性属于实例(对象),只能通过对象名访问;而类属性属于类,可以通过类名或对象名访问。

在类的方法中可以调用类本身的其他方法,也可以访问类属性以及对象属性。在Python 中比较特殊的是,可以动态地为类和对象增加成员,这一点是和很多面向对象程序设计语言不同的,也是 Python 动态类型特点的一种重要体现。

```
class Car:
    price=100000           #定义类属性
    def __init__(self, c):
        self.color=c       #定义实例属性

car1=Car("Red")
car2=Car("Blue")
print(car1.color, Car.price)
Car.price=110000           #修改类属性
Car.name='QQ'              #增加类属性
car1.color="Yellow"        #修改实例属性
print(car2.color, Car.price, Car.name)
print(car1.color, Car.price, Car.name)
```

### 6.1.4 私有成员与公有成员

Python 并没有对私有成员提供严格的访问保护机制。在定义类的属性时,如果属性名以两个下划线“__”(中间无空)开头则表示是私有属性。私有属性在类的外部不能直接访问,需要通过调用对象的公有成员方法来访问,或者通过 Python 支持的特殊方式来访问。Python 提供了访问私有属性的特殊方式,可用于程序的测试和调试,对于成员方法也具有同样的性质。

私有属性是为了数据封装和保密而设的属性,一般只能在类的成员方法(类的内部)中使用访问,虽然 Python 支持一种特殊的方式来从外部直接访问类的私有成员,但是并不推荐这样做。公有属性是可以公开使用的,既可以在类的内部进行访问,也可以在外部程序中使用。

```
>>> class A:
    def __init__(self, value1=0, value2=0):
        self._value1=value1
        self.__value2=value2
    def setValue(self, value1, value2):
        self._value1=value1
```

```
        self.__value2=value2
    def show(self):
        print(self._value1)
        print(self.__value2)
>>> a=A()
>>> a._value1
0
>>> a._A__value2            #在外部访问对象的私有数据成员
0
```

在 IDLE 环境中，在对象或类名后面加上一个圆点“.”，稍等一秒钟则会自动列出其所有公开成员，例如图 6-1 所示，模块也具有同样的特点。

而如果在圆点“.”后面再加一个下划线，则会列出该对象或类的所有成员，包括私有成员，如图 6-2 所示。

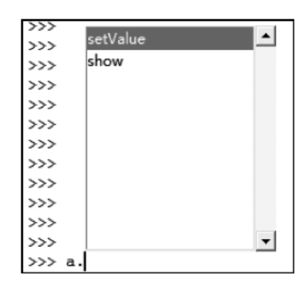



图 6-1 列出对象公开成员

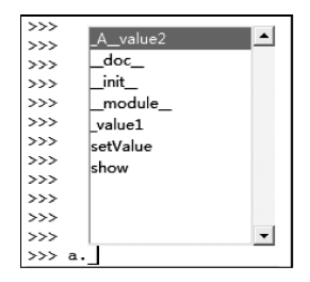



图 6-2 列出对象所有成员

在 Python 中，以下划线开头的变量名和方法名有特殊的含义，尤其是在类的定义中。用下划线作为变量名和方法名前缀和后缀来表示类的特殊成员。

- _xxx：这样的对象叫做保护成员，不能用“from module import *”导入，只有类对象和子类对象能访问这些成员；
- __xxx__：系统定义的特殊成员；
- __xxx：类中的私有成员，只有类对象自己能访问，子类对象也不能访问到这个成员，但在对象外部可以通过“对象名._类名__xxx”这样的特殊方式来访问。Python 中不存在严格意义上的私有成员。

另外，在 IDLE 交互模式下，一个下划线“_”表示解释器中最后一次显示的内容或最后一次语句正确执行的输出结果。例如：

```
>>> 3+5
8
>>> _+2
10
>>> _ * 3
30
>>> _ / 5
6
```

```
>>> 3
3
>>> _
3
>>> 1/0

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in<module>
    1/0
ZeroDivisionError: integer division or modulo by zero
>>> _
3
```

下面的代码演示了特殊成员定义和访问的方法。

```
>>> class Fruit:
        def __init__(self):
            self.__color='Red'
            self.price=1
>>> apple=Fruit()
>>> apple.price                               #显示对象公开数据成员的值
1
>>> apple.price=2                             #修改对象公开数据成员的值
>>> apple.price
2
>>> print(apple.price, apple._Fruit__color)  #显示对象私有数据成员的值
2 Red
>>> apple._Fruit__color="Blue"                #修改对象私有数据成员的值
>>> print(apple.price, apple._Fruit__color)
2 Blue
>>> print(apple.__color)                     #不能直接访问对象的私有数据成员，出错
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in<module>
    print(apple.__color )
AttributeError: Fruit instance has no attribute '__color'
>>> peach=Fruit()
>>> print(peach.price, peach._Fruit__color)
1 Red
```

## 6.2 方法

在类中定义的方法可以粗略分为四大类：公有方法、私有方法、静态方法和类方法。其中，公有方法、私有方法都属于对象，私有方法的名字以两个下划线“__”开始，每个对象都有自己的公有方法和私有方法，在这两类方法中可以访问属于类和对象的成员；公有方

法通过对象名直接调用,私有方法不能通过对象名直接调用,只能在属于对象的方法中通过 self 调用或在外部通过 Python 支持的特殊方式来调用。如果通过类名来调用属于对象的公有方法,需要显式为该方法的 self 参数传递一个对象名,用来明确指定访问哪个对象的数据成员。静态方法和类方法都可以通过类名和对象名调用,但不能直接访问属于对象的成员,只能访问属于类的成员。一般将 cls 作为类方法的第一个参数名称,但也可以使用其他的名字作为参数,并且在调用类方法时不需要为该参数传递值。例如,下面的代码所演示:

```
>>> class Root:
    __total=0
    def __init__(self, v):
            self.__value=v
            Root.__total+=1

    def show(self):
            print('self.__value:', self.__value)
            print('Root.__total:', Root.__total)

    @classmethod
    def classShowTotal(cls):      #类方法
            print(cls.__total)

    @staticmethod
    def staticShowTotal():        #静态方法
            print(Root.__total)
>>> r=Root(3)
>>> r.classShowTotal()            #通过对象来调用类方法
1
>>> r.staticShowTotal()           #通过对象来调用静态方法
1
>>> r.show()
self.__value: 3
Root.__total: 1
>>> rr=Root(5)
>>> Root.classShowTotal()         #通过类名调用类方法
2
>>> Root.staticShowTotal()        #通过类名调用静态方法
2
>>> Root.show()                   #试图通过类名直接调用实例方法,失败
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in<module>
    Root.show()
TypeError: unbound method show() must be called with Root instance as first
```

```
argument (got nothing instead)
>>> Root.show(r)                    #但是可以通过这种方法来调用方法并访问实例成员
self.__value: 3
Root.__total: 2
>>> r.show()
self.__value: 3
Root.__total: 2
>>> Root.show(rr)                   #通过类名调用实例方法时为 self 参数显式传递对象名
self.__value: 5
Root.__total: 2
>>> rr.show()
self.__value: 5
Root.__total: 2
```

## 6.3 属性

Python 2.x 和 Python 3.x 对属性的实现和处理方式不一样，内部实现有较大的差异，使用时应注意二者之间的区别。需要注意的是，本节中讨论的“属性”指狭义的概念，与前面所谈的“属性”概念并不完全一样。

### 6.3.1 Python 2.x 中的属性

在 Python 2.x 中，使用@property 或 property()函数来声明一个属性，然而属性并没有得到真正意义的实现，也没有提供应有的访问保护机制。正如前面所说，在 Python 中，可以为类和对象动态增加新成员。在 Python 2.x 中，为对象增加新的数据成员时，将隐藏同名的已有属性。例如，下面的 Python 2.7.8 代码：

```
>>> class Test:
    def __init__(self, value):
            self.__value=value

    @property
        def value(self):
            return self.__value
>>> a=Test(3)
>>> a.value
3
>>> a.value=5                       #动态添加了新成员，隐藏了定义的属性
>>> a.value
5
>>> t._Test__value                  #原来的私有变量没有改变
3
```

除了动态增加成员时会隐藏已有属性，下面的代码从表面看来是修改属性的值，而实际上也是增加了新成员，从而隐藏了已有属性。

```
>>> class Test:
    def __init__(self, value):
        self.__value=value

    def __get(self):
        return self.__value

    def __set(self, v):
        self.__value=v

    value=property(__get, __set)

    def show(self):
        print self.__value
>>> t=Test(3)
>>> t.value
3
>>> t.value+=2              #动态添加了新成员
>>> t.value                 #这里访问的是新成员
5
>>> t.show()                #访问原来定义的私有数据成员
3
>>> del t.value             #这里删除的是刚才添加的新成员
>>> t.value                 #访问原来的属性
3
>>> del t.value             #试图删除属性
出错信息(略)
AttributeError: Test instance has no attribute 'value'
>>> del t._Test__value      #删除私有成员
>>> t.value                 #访问属性，但该属性对应的私有成员已不存在
出错信息(略)
AttributeError: Test instance has no attribute '_Test__value'
```

下面的代码则更加清楚地演示了 Python 2.x 中私有成员和普通成员之间的关系，有助于理解上面的内容。

```
>>> class Test:
    def show(self):
        print self.value
        print self.__v
>>> t=Test()
>>> t.show()
```

```
出错信息(略)
AttributeError: Test instance has no attribute 'value'
>>> t.value=3
>>> t.show()
3
出错信息(略)
AttributeError: Test instance has no attribute '_Test__v'
>>> t.__v=5
>>> t.show()
3
出错信息(略)
AttributeError: Test instance has no attribute '_Test__v'
>>> t._Test__v=5
>>> t.show()
3
5
```

### 6.3.2 Python 3.x 中的属性

在 Python 3.x 中,属性得到了较为完整的实现,支持更加全面的保护机制。例如下面的代码所示,如果设置属性为只读,则无法修改其值,也无法为对象增加与属性同名的新成员,同时,也无法删除对象属性。例如,下面的代码运行在 Python 3.4.2 中:

```
>>> class Test:
    def __init__(self, value):
        self.__value=value

    @property
    def value(self):            #只读,无法修改和删除
        return self.__value
>>> t=Test(3)
>>> t.value
3
>>> t.value=5                   #只读属性不允许修改值
出错信息(略)
AttributeError: can't set attribute
>>> t.v=5                       #动态增加新成员
>>> t.v
5
>>> del t.v                     #动态删除成员
>>> del t.value                 #试图删除对象属性,失败
出错信息(略)
AttributeError: can't delete attribute
```

```
>>> t.value
3
```

下面的代码则把属性设置为可读、可修改，而不允许删除。

```
>>> class Test:
    def __init__(self, value):
        self.__value=value

    def __get(self):
        return self.__value

    def __set(self, v):
        self.__value=v

    value=property(__get, __set)

    def show(self):
        print(self.__value)
>>> t=Test(3)
>>> t.value                    #允许读取属性值
3
>>> t.value=5                  #允许修改属性值
>>> t.value
5
>>> t.show()                   #属性对应的私有变量也得到了相应的修改
5
>>> del t.value                #试图删除属性，失败
出错信息
AttributeError: can't delete attribute
```

当然，也可以将属性设置为可读、可修改、可删除。

```
>>> class Test:
    def __init__(self, value):
        self.__value=value

    def __get(self):
        return self.__value

    def __set(self, v):
        self.__value=v

    def __del(self):
        del self.__value
```

```
    value=property(__get, __set, __del)

    def show(self):
        print(self.__value)
>>> t=Test(3)
>>> t.show()
3
>>> t.value
3
>>> t.value=5
>>> t.show()
5
>>> t.value
5
>>> del t.value
>>> t.value
出错信息(略)
AttributeError: 'Test' object has no attribute '_Test__value'
>>> t.show()
出错信息(略)
AttributeError: 'Test' object has no attribute '_Test__value'
>>> t.value=1          #为对象动态增加属性和对应的私有数据成员
>>> t.show()
1
>>> t.value
1
```

## 6.4 特殊方法与运算符重载

### 6.4.1 常用特殊方法

Python 类有大量的特殊方法,其中比较常见的是构造函数和析构函数。Python 中类的构造函数是__init__(),一般用来为数据成员设置初值或进行其他必要的初始化工作,在创建对象时被自动调用和执行,可以通过为构造函数定义默认值参数来实现类似于其他语言中构造函数重载的目的。如果用户没有设计构造函数,Python 将提供一个默认的构造函数用来进行必要的初始化工作。Python 中类的析构函数是__del__(),一般用来释放对象占用的资源,在 Python 删除对象和收回对象空间时被自动调用和执行。如果用户没有编写析构函数,Python 将提供一个默认的析构函数进行必要的清理工作。

在 Python 中,除了构造函数和析构函数之外,还有大量的特殊方法支持更多的功能,例如运算符重载就是通过在类中重写特殊函数来实现的。表 6-1 列出了其中一部分特殊方法。

表 6-1　Python 类特殊方法

| 方　法 | 功能说明 |
| --- | --- |
| __init__() | 构造函数，生成对象时调用 |
| __del__() | 析构函数，释放对象时调用 |
| __add__()、__radd__() | 左+、右+ |
| __sub__() | - |
| __mul__() | * |
| __div__()、__truediv__() | /，Python 2.x 使用__div__()，Python 3.x 使用__truediv__() |
| __floordiv__() | 整除 |
| __mod__() | % |
| __pow__() | ** |
| __cmp__() | 比较运算 |
| __repr__() | 打印、转换 |
| __setitem__() | 按照索引赋值 |
| __getitem__() | 按照索引获取值 |
| __len__() | 计算长度 |
| __call__() | 函数调用 |
| __contains__() | 测试是否包含某个元素 |
| __eq__()、__ne__()、__lt__()、__le__()、__gt__()、__ge__() | ==、!=、<、<=、>、>= |
| __str__() | 转化为字符串 |
| __lshift__()、__rshift__() | <<、>> |
| __and__()、__or__()、__invert__() | &、\|、~ |
| __iadd__()、__isub__() | +=、-= |

## 6.4.2　案例精选

下面通过一个示例来演示特殊方法的用法。在下面的代码中，定义了一个数组类，重写了一部分特殊方法以支持数组之间、数组与整数之间的四则运算以及内积、大小比较、成员测试和元素访问等运算符。代码使用 Python 2.7.8 编写，可以很容易地改写为 Python 3.x 的版本。

```
#Filename: MyArray.py
#Function description: Array and its operating
#--------------------
```

```
import types

class MyArray:
    '''All the elements in this array must be numbers'''
    __value=[]
    __size=0

    def __IsNumber(self, n):
        if type(n) !=types.ComplexType and type(n) !=types.FloatType \
            and type(n) !=types.IntType and type(n) !=types.LongType:
            return False
        return True

    def __init__(self, *args):
        for arg in args:
            if not self.__IsNumber(arg):
                print 'All elements must be numbers'
                return
        self.__value=[]
        for arg in args:
            self.__value.append(arg)
        self.__size=len(args)

    def __add__(self, n):
        if not self.__IsNumber(n):
            print '+operating with ', type(n), ' and number type is not supported.'
            return
        b=MyArray()
        for v in self.__value:
            b.__value.append(v+n)
        return b

    def __sub__(self, n):
        if not self.__IsNumber(n):
            print '-operating with ', type(n), ' and number type is not supported.'
            return
        b=MyArray()
        for v in self.__value:
            b.__value.append(v-n)
        return b

    def __mul__(self, n):
        if not self.__IsNumber(n):
```

```
            print '* operating with',type(n),'and number type is not supported.'
            return
        b=MyArray()
        for v in self.__value:
            b.__value.append(v * n)
        return b

    def __div__(self, n):
        if not self.__IsNumber(n):
            print r'/operating with ',type(n),' and number type is not supported.'
            return
        if type(n)==types.IntType:
            n=float(n)
        b=MyArray()
        for v in self.__value:
            b.__value.append(v / n)
        return b

    def __mod__(self, n):
        if not self.__IsNumber(n):
            print r'%operating with ', type(n), ' and number type is not supported.'
            return
        b=MyArray()
        for v in self.__value:
            b.__value.append(v %n)
        return b

    def __pow__(self, n):
        if not self.__IsNumber(n):
            print '** operating with',type(n),'and number type is not supported.'
            return
        b=MyArray()
        for v in self.__value:
            b.__value.append(v ** n)
        return b

    def __len__(self):
        return len(self.__value)

    #for: x
    #when use the object as a statement directly, the function will be called
    def __repr__(self):
        #equivalent to return 'self.__value'
        return repr(self.__value)
```

```
#for: print x
def __str__(self):
    return str(self.__value)

def append(self, v):
    if not self.__IsNumber(v):
        print 'Only number can be appended.'
        return
    self.__value.append(v)
    self.__size+=1

def __getitem__(self, index):
    if self.__IsNumber(index) and 0<=index<=self.__size:
        return self.__value[index]
    else:
        print 'Index out of range.'

def __setitem__(self, index, v):
    if self.__IsNumber(index) and 0<=index<=self.__size:
        if self.__IsNumber(v):
            self.__value[index]=v
        else:
            print v, ' is not a number'
    else:
        print index, ' is not a number or out of range.'

#member test. support the keyword 'in'
def __contains__(self, v):
    if v in self.__value:
        return True
    return False

#dot product
def dot(self, v):
    if not isinstance(v, MyArray):
        print v, ' must be an instance of MyArray.'
        return
    if len(v) !=self.__size:
        print 'The size must be equal.'
        return
    b=MyArray()
    for m, n in zip(v.__value, self.__value):
        b.__value.append(m * n)
```

```
        return sum(b.__value)

    #equal to
    def __eq__(self, v):
        if not isinstance(v, MyArray):
            print v, ' must be an instance of MyArray.'
            return
        if cmp(self.__value, v.__value)==0:
            return True
        return False

    #less than
    def __lt__(self, v):
        if not isinstance(v, MyArray):
            print v, ' must be an instance of MyArray.'
            return
        if cmp(self.__value, v.__value)<0: #在 Python 3.x 中可以直接使用 self__
value<v.__value
            return True
        return False

if __name__=='__main__':
    print 'Please use me as a module.'
```

将上面的代码保存为 MyArray.py 文件之后，将其作为模块导入来使用，下面的代码简单演示了 MyArray 类的用法。

```
>>> import MyArray
>>> a=MyArray.MyArray(1, 2, 3, 4, 5, 6)
>>> b=MyArray.MyArray(6, 5, 4, 3, 2, 1)
>>> len(a)
6
>>> a.dot(b)
56
>>> a<b
True
>>> a>b
False
>>> a==a
True
>>> 3 in a
True
>>> a * 3
[3, 6, 9, 12, 15, 18]
>>> a+2
```

```
[3, 4, 5, 6, 7, 8]
>>> a ** 2
[1, 4, 9, 16, 25, 36]
>>> a / 2
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
>>> a
[1, 2, 3, 4, 5, 6]
>>> a[0]=8
>>> a
[8, 2, 3, 4, 5, 6]
```

## 6.5 继承机制

继承是为代码复用和设计复用而设计的，是面向对象程序设计的重要特性之一。当设计一个新类时，如果可以继承一个已有的设计良好的类然后进行二次开发，无疑会大幅度减少开发工作量。在继承关系中，已有的、设计好的类称为父类或基类，新设计的类称为子类或派生类。派生类可以继承父类的公有成员，但是不能继承其私有成员。如果需要在派生类中调用基类的方法，可以使用内置函数 super()或者通过“基类名.方法名()”的方式来实现这一目的。

Python 支持多继承，如果父类中有相同的方法名，而在子类中使用时没有指定父类名，则 Python 解释器将从左向右按顺序进行搜索。

**例 6-1** 设计 Person 类，并根据 Person 派生 Teacher 类，分别创建 Person 类与 Teacher 类的对象。

```
import types
class Person(object): #基类必须继承于 object,否则在派生类中将无法使用 super()函数
    def __init__(self, name='', age=20, sex='man'):
        self.setName(name)
        self.setAge(age)
        self.setSex(sex)

    def setName(self, name):
        if type(name) !=types.StringType:
            print 'name must be string.'
            return
        self.__name=name

    def setAge(self, age):
        if type(age) !=types.IntType:
            print 'age must be integer.'
            return
        self.__age=age
```

```
    def setSex(self, sex):
        if sex !='man' and sex !='woman':
            print 'sex must be "man" or "woman"'
            return
        self.__sex=sex

    def show(self):
        print self.__name
        print self.__age
        print self.__sex

class Teacher(Person):
    def __init__(self, name='', age=30, sex='man', department='Computer'):
        #调用基类构造方法初始化基类的私有数据成员
        super(Teacher, self).__init__(name, age, sex)
        #Person.__init__(self, name, age, sex) #也可以这样初始化基类的私有数据成员
        self.setDepartment(department)        #初始化派生类的数据成员

    def setDepartment(self, department):
        if type(department) !=types.StringType:
            print 'department must be a string.'
            return
        self.__department=department

    def show(self):
        super(Teacher, self).show()
        print self.__department

if __name__=='__main__':
    zhangsan=Person('Zhang San', 19, 'man')
    zhangsan.show()
    lisi=Teacher('Li Si', 32, 'man', 'Math')
    lisi.show()
    lisi.setAge(40)                           #调用继承的方法修改年龄
    lisi.show()
```

为了更好地理解 Python 类的继承机制，我们来看下面的 Python 2.7.8 代码，请认真体会构造函数、私有方法以及普通公开方法的继承原理。

```
>>> class A():
        def __init__(self):
            self.__private()
            self.public()
```

```
        def __private(self):
                print '__private() method of A'

        def public(self):
                print 'public() method of A'
>>> class B(A):
        def __private(self):
                print '__private() method of B'

        def public(self):
                print 'public() method of B'
>>> b=B()                              #自动调用从基类 A 继承的构造函数
__private() method of A
public() method of B
>>> print '\n'.join(dir(b))           #查看对象 b 的成员
_A__private
_B__private
__doc__
__init__
__module__
Public
>>> class C(A):
    def __init__(self):
        self.__private()
        self.public()

    def __private(self):
        print '__private() method of C'

    def public(self):
        print 'public() method of C'
>>> c=C()                              #自动调用派生类自己的构造函数
__private() method of C
public() method of C
>>> print '\n'.join(dir(c))
_A__private
_C__private
__doc__
__init__
__module__
public
```

## 本章小结

(1) 面向对象程序设计(Object Oriented Programming,OOP)的思想主要针对大型软件设计而提出,使得软件设计更加灵活,能够很好地支持代码复用和设计复用,并且使得代码具有更好的可读性和可扩展性。

(2) 定义类时使用关键字 class。

(3) 可以动态地为类和对象增加成员。

(4) 类中所有实例方法都至少包含一个 self 参数,并且必须是第一个参数,用来表示对象本身,通过对象名调用实例方法时不需要为 self 参数传递任何值。

(5) 实例属性一般是指在构造函数__init__()中定义的,定义时以 self 作为前缀;类属性是在类中所有方法之外定义的数据成员。

(6) 如果通过类名来调用属于对象的公有方法,需要显式为该方法的 self 参数传递一个对象名,用来明确指定访问哪个对象的数据成员。

(7) Python 2.x 中的属性没有提供完整的保护机制,在 Python 3.x 中得到了完整的实现。

(8) 在 Python 中,运算符重载是通过重新实现一些特殊函数来实现的。

(9) Python 支持多继承,如果多个父类中有相同名字的成员,而在子类中使用该成员时没有指定其所属父类名,则 Python 解释器将从左向右按顺序进行搜索。

(10) 在 Python 中,以下划线开头的变量名有特殊的含义,尤其是在类的定义中。

(11) 在 IDLE 交互式环境中,单个下划线表示上次语句正常执行的输出结果。

## 习题

6.1 继承 6.5 节例 6-2 中的 Person 类生成 Student 类,编写新的方法用来设置学生专业,然后生成该类对象并显示信息。

6.2 设计一个三维向量类,并实现向量的加法、减法以及向量与标量的乘法和除法运算。

6.3 面向对象程序设计的三要素分别为________、________和________。

6.4 简单解释 Python 中以下划线开头的变量名特点。

6.5 与运算符“**”对应的特殊方法名为________,与运算符“//”对应的特殊方法名为________。

6.6 假设 a 为类 A 的对象且包含一个私有数据成员“__value”,那么在类的外部通过对象 a 直接将其私有数据成员“__value”的值设置为 3 的语句可以写作________。

# 第 7 章　文 件 操 作

为了长期保存数据以便重复使用、修改和共享，必须将数据以文件的形式存储到外部存储介质(如磁盘、U 盘、光盘等)或云盘中。管理信息系统是使用数据库来存储数据的，而数据库最终还是要以文件的形式存储到硬盘或其他存储介质上，应用程序的配置信息往往也是使用文件来存储的，图形、图像、音频、视频、可执行文件等等也都是以文件的形式存储在磁盘上的。因此，文件操作在各类应用软件的开发中均占有重要的地位。

按文件中数据的组织形式可以把文件分为文本文件和二进制文件两大类。

**1. 文本文件**

文本文件存储的是常规字符串，由若干文本行组成，通常每行以换行符'\n'结尾。常规字符串是指记事本或其他文本编辑器能正常显示、编辑并且人类能够直接阅读和理解的字符串，如英文字母、汉字、数字字符串。文本文件可以使用字处理软件，如 gedit、记事本进行编辑。

**2. 二进制文件**

二进制文件把对象内容以字节串(bytes)进行存储，无法用记事本或其他普通文本处理软件直接进行编辑，通常也无法被人类直接阅读和理解，需要使用专门的软件进行解码后读取、显示、修改或执行。常见的如图形图像文件、音视频文件、可执行文件、资源文件、各种数据库文件、各类 Office 文档等都属于二进制文件。

## 7.1　文件对象

无论是文本文件还是二进制文件，其操作流程基本都是一致的，即：首先打开文件并创建文件对象，然后通过该文件对象对文件内容进行读取、写入、删除、修改等操作，最后关闭并保存文件内容。Python 内置了文件对象，通过 open()函数即可以指定模式打开指定文件并创建文件对象，例如：

```
文件对象名=open(文件名[, 打开方式[, 缓冲区]])
```

其中，文件名指定了被打开的文件名称，如果要打开的文件不在当前目录中，还需要指定完整路径，为了减少完整路径中“\”符号的输入，可以使用原始字符串；打开模式(见表 7-1)指定了打开文件后的处理方式，例如“只读”、“读写”、“追加”等等；缓冲区指定了读写文件的缓存模式，数值 0 表示不缓存，数值 1 表示缓存，如大于 1 则表示缓冲区的大小，默认值是缓存模式。如果执行正常，open()函数返回 1 个文件对象，通过该文件对象

可以对文件进行各种操作，如果指定文件不存在、访问权限不够、磁盘空间不够或其他原因导致创建文件对象失败则抛出异常。例如，下面的代码分别以读、写方式打开了两个文件并创建了与之对应的文件对象。

```
f1=open( 'file1.txt', 'r' )
f2=open( 'file2.txt', 'w')
```

当对文件内容操作完以后，一定要关闭文件，以保证所做的任何修改都得到保存。

```
f1.close()
```

文件对象常用属性如表 7-2 所示。

**表 7-1　文件打开模式**

| 模式 | 说　　明 |
| --- | --- |
| r | 读模式 |
| w | 写模式 |
| a | 追加模式 |
| b | 二进制模式(可与其他模式组合使用) |
| + | 读、写模式(可与其他模式组合使用) |

**表 7-2　文件对象属性**

| 属　性 | 说　　明 |
| --- | --- |
| closed | 判断文件是否关闭，若文件被关闭，则返回 True |
| mode | 返回文件的打开模式 |
| name | 返回文件的名称 |

文件对象常用方法如表 7-3 所示。

**表 7-3　文件对象常用方法**

| 方　　法 | 功 能 说 明 |
| --- | --- |
| flush() | 把缓冲区的内容写入文件，但不关闭文件 |
| close() | 把缓冲区的内容写入文件，同时关闭文件，并释放文件对象 |
| read([size]) | 从文件中读取 size 个字节(Python 2. x)或字符(Python 3. x)的内容作为结果返回，如果省略 size，则表示一次性读取所有内容 |
| readline() | 从文本文件中读取一行内容作为结果返回 |
| readlines() | 把文本文件中的每行文本作为一个字符串存入列表中，返回该列表 |
| seek(offset[, whence]) | 把文件指针移动到新的位置，offset 表示相对于 whence 的位置。whence 为 0 表示从文件头开始计算，1 表示从当前位置开始计算，2 表示从文件尾开始计算，默认为 0 |
| tell() | 返回文件指针的当前位置 |
| truncate([size]) | 删除从当前指针位置到文件末尾的内容。如果指定了 size，则不论指针在什么位置都只留下前 size 个字节，其余的删除 |
| write(s) | 把字符串 s 的内容写入文件 |
| writelines(s) | 把字符串列表写入文本文件，不添加换行符 |

## 7.2 文本文件操作案例精选

在本节中，主要通过几个示例来演示文本文件的读写操作。对于 read()、write()以及其他读写方法，当读写操作完成之后，都会自动移动文件指针，如果需要对文件指针进行定位，可以使用 seek()方法，如果需要获知文件指针当前位置可以使用 tell()方法。

**例 7-1** 向文本文件中写入内容。

```
f=open('sample.txt', 'a+')
s='文本文件的读取方法\n文本文件的写入方法\n'
f.write(s)
f.close()
```

对于上面的代码，建议写为如下形式：

```
s='文本文件的读取方法\n文本文件的写入方法\n'
with open('sample.txt', 'a+') as f:
    f.write(s)
```

使用上下文管理关键字 with 可以自动管理资源，不论何种原因跳出 with 块，总能保证文件被正确关闭，并且可以在代码块执行完毕后自动还原进入该代码块时的现场。

**例 7-2** 读取并显示文本文件的前 5 个字节。

对于文件对象的 read()方法，Python 2.x 和 Python 3.x 的解释略有不同，尤其是文本文件中包含中文的时候。Python 2.x 的 read()方法是读取文件中指定数量的字节，对于中文可能会由于无法正常解码而出现乱码。例如，假设 sample.txt 文件内容为“SDIBT 中国山东烟台”，那么在 Python 2.7.8 中代码运行结果如下：

```
>>> fp=open( 'sample.txt', 'r')
>>> print fp.read(5)
SDIBT
>>> print fp.read(7)
Öйú□
>>> print fp.read(8)
□NÌ
>>> fp.close()
```

而 Python 3.x 对中文支持较好，对 read()方法的解释是读取文件中指定数量的字符而不是字节，对中文和英文字母同等对待。对前述 sample.txt 文件，Python 3.4.2 中代码运行结果如下：

```
>>> fp=open('sample.txt', 'r')
>>> print(fp.read(5))
SDIBT
>>> print(fp.read(7))
中国山东烟台
```

```
>>> fp.seek(0)
0
>>> print(fp.read(8))
SDIBT 中国山
```

**例 7-3** 读取并显示文本文件所有行。

```
f=open('sample.txt', 'r')
while True:
    line=f.readline()
    if line=='':
        break
    print line,          #逗号不会产生换行符,但文件中有换行符,因此会换行
f.close()
```

当然,也可以写作:

```
f=open(sample.txt', 'r')
li=f.readlines()
for line in li:
    print line,
f.close()
```

**例 7-4** 移动文件指针。

Python 2.x 和 Python 3.x 对于 seek()方法的理解和处理是一致的,即将文件指针定位到文件中指定字节的位置。但是由于对中文的支持程度不一样,可能会导致在 Python 2.x 和 Python 3.x 中的运行结果有所不同。例如,下面的代码在 Python 3.4.2 中运行,当遇到无法解码的字符会抛出异常。

```
>>> s='中国山东烟台 SDIBT'
>>> fp=open(r'D:\sample.txt', 'w')
>>> fp.write(s)
11
>>> fp.close()
>>> fp=open(r'D:\sample.txt', 'r')
>>> print(fp.read(3))
中国山
>>> fp.seek(2)
2
>>> print(fp.read(1))
国
>>> fp.seek(13)
13
>>> print(fp.read(1))
D
>>> fp.seek(15)
```

```
15
>>> print(fp.read(1))
B
>>> fp.seek(3)
3
>>> print(fp.read(1))
出错信息
UnicodeDecodeError: 'gbk' codec can't decode byte 0xfa in position 0: illegal
multibyte sequence
```

而在 Python 2.7.8 中,则不抛出异常,而是输出乱码,例如下面的代码:

```
>>> s='中国山东烟台 SDIBT'
>>> fp=open(r'D:\sample.txt', 'w')
>>> fp.write(s)
>>> fp.close()
>>> fp=open(r'D:\sample.txt', 'r')
>>> print(fp.read(3))
Öй
>>> fp.seek(2)
>>> print(fp.read(3))
□úÉ
>>> print(fp.read(2))
蕉
```

**例 7-5** 读取文本文件 data.txt 中所有整数,将其按升序排序后再写入文本文件 data_asc.txt 中。

```
with open('data.txt', 'r') as fp:
    data=fp.readlines()
data=[int(line.strip()) for line in data]
data.sort()
data=[str(i)+'\n' for i in data]
with open('data_asc.txt', 'w') as fp:
    fp.writelines(data)
```

**例 7-6** 编写程序,保存为 demo6.py,运行后生成文件 demo6_new.py,其中的内容与 demo6.py 一致,但是在每行的行尾加上了行号。

```
filename='demo6.py'
with open(filename, 'r') as fp:
    lines=fp.readlines()
lines=[line.rstrip()+' '*(100-len(line))+'#'+str(index)+'\n' for index,
line in enumerate(lines)]
with open(filename[:-3]+'_new.py', 'w') as fp:
    fp.writelines(lines)
```

**例 7-7** Python 程序中代码复用度检测。

```
#-*-coding:utf-8-*-
#Filename: codeReusecheck\FindLongestReuse.py
#--------------------
#Function description:Find the longest matches in source codes
#--------------------
#Author: Dong Fuguo
#QQ: 306467355
#Email: dongfuguo2005@126.com
#--------------------
#Date: 2014-11-16
#--------------------

from os.path import isfile as isfile
from time import time as time

Result={}
AllLines=[]
FileName=r'FindLongestReuse.py'
#FileName=input('Please input the file to check, including full path:')

#Read the content of given file
#Remove blank lines
#Remove all the whitespace string  of every line,
#preserving only one space character between words or operators
#note:The last line does not contain the '\n' character
def PreOperate():
    global AllLines
    with open(FileName, 'r') as fp:
        for line in fp:
            line=' '.join(line.split())
            if line !='':
                AllLines.append(line)

#Check if the current position is still the duplicated one
def IfHasDuplicated(Index1):
    for item in Result.values():
        for it in item:
            if Index1==it[0]:
                return it[1]        #return the span
    return False

#If the current line Index2 is in a span of duplicated lines, return True,
```

```
else False
def IsInSpan(Index2):
    for item in Result.values():
        for i in item:
            if i[0]<=Index2<i[0]+i[1]:
                return True
    return False

def MainCheck():
    global Result
    TotalLen=len(AllLines)
    Index1=0
    while Index1<TotalLen-1:
        #speed up
        span=IfHasDuplicated(Index1)
        if span:
            Index1+=span
            continue
        Index2=Index1+1
        while Index2<TotalLen:
            #speed up, skip the duplicated lines
            if IsInSpan(Index2):
                Index2+=1
                continue
            src=''
            des=''
            for i in range(10):
                if Index2+i>=TotalLen:
                    break
                src+=AllLines[Index1+i]
                des+=AllLines[Index2+i]
                if src==des:
                    t=Result.get(Index1, [])
                    for tt in t:
                        if tt[0]==Index2:
                            tt[1]=i+1
                            break
                    else:
                        t.append([Index2, i+1])
                    Result[Index1]=t
                else:
                    break
            t=Result.get(Index1, [])
            for tt in t:
```

```
                    if tt[0]==Index2:
                        Index2+=tt[1]
                        break
                else:
                    Index2+=1

            #Optimize the Result dictionary, remove the items with span<3
            Result[Index1]=Result.get(Index1, [])
            for n in Result[Index1][::-1]: #Note: here must use the reverse slice,
                if n[1]<3:
                    Result[Index1].remove(n)
            if not Result[Index1]:
                del Result[Index1]

            #Compute the min span of duplicated codes of line Index1,modify the step
            Index1
            a=[ttt[1] for ttt in Result.get(Index1, [[Index1, 1]])]
            if a:
                Index1+=max(a)
            else:
                Index1+=1

#Output the result
def Output():
    print '-'*20
    #print 'The PreOperated text is:'
    #print AllLines
    print '-'*20
    print 'Result:'
    for key, value in Result.items():
        print 'The original line is: \n {0}'.format(AllLines[key])
        print 'Its line number is {0}'.format(key)
        print 'The duplicated line numbers are:'
        for i in value:
            print '    Start:', i[0], '    Span:', i[1]
        print '-'*20
    print '-'*20

if isfile(FileName):
    start=time()
    PreOperate()
    MainCheck()
    Output()
    print 'Time used:', time()-start
```

## 7.3 二进制文件操作案例精选

数据库文件、图像文件、可执行文件、音视频文件、Office 文档等等均属于二进制文件。对于二进制文件，不能使用记事本或其他文本编辑软件进行正常读写，也无法通过 Python 的文件对象直接读取和理解二进制文件的内容。必须正确理解二进制文件结构和序列化规则，才能准确地理解二进制文件内容并且设计正确的反序列化规则。所谓序列化，简单地说就是把内存中的数据在不丢失其类型信息的情况下转成对象的二进制形式的过程，对象序列化后的形式经过正确的反序列化过程应该能够准确无误地恢复为原来的对象。

Python 中常用的序列化模块有 struct、pickle、json、marshal 和 shelve，其中 pickle 有 C 语言实现的 cPickle，速度约提高 1000 倍，应优先考虑使用。本节主要介绍 struct 和 pickle 模块在对象序列化和二进制文件操作方面的应用，其他模块请参考有关文档。

### 7.3.1 使用 pickle 模块

pickle 是较为常用并且速度非常快的二进制文件序列化模块，下面通过两个示例来了解一下如何使用 pickle 模块进行对象序列化和二进制文件读写。

**例 7-8** 使用 pickle 模块写入二进制文件。

```
import pickle

f=open('sample_pickle.dat', 'wb')
n=7
i=13000000
a=99.056
s='中国人民 123abc'
lst=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
tu=(-5, 10, 8)
coll={4, 5, 6}
dic={'a':'apple', 'b':'banana', 'g':'grape', 'o':'orange'}
try:
    pickle.dump(n, f)              #表示后面将要写入的数据个数
    pickle.dump(i, f)              #把整数 i 转换为字节串,并写入文件
    pickle.dump(a, f)
    pickle.dump(s, f)
    pickle.dump(lst, f)
    pickle.dump(tu, f)
    pickle.dump(coll, f)
    pickle.dump(dic, f)
```

```
except:
    print '写文件异常!'              #如果写文件异常则跳到此处执行
finally:
    f.close()
```

**例 7-9** 读取例 7-8 中写入二进制文件的内容。

```
import pickle

f=open('sample_pickle.dat', 'rb')
n=pickle.load(f)                    #读出文件的数据个数
i=0
while i<n:
    x=pickle.load(f)
    print x
    i=i+1
f.close()
```

### 7.3.2 使用 struct 模块

struct 也是比较常用的对象序列化和二进制文件读写模块,下面通过两个示例来简单介绍使用 struct 模块对二进制文件进行读写的用法。

**例 7-10** 使用 struct 模块写入二进制文件。

```
import struct

n=1300000000
x=96.45
b=True
s='a1@中国'
sn=struct.pack('if?', n, x, b)    #把整数 n、浮点数 x、布尔对象 b 依次转换为字节串
f=open('sample_struct.dat', 'wb')
f.write(sn)                         #写入字节串
f.write(s)                          #字符串可直接写入
f.close()
```

**例 7-11** 使用 struct 模块读取例 7-10 写入二进制文件的内容。

```
import struct

f=open('sample_struct.dat', 'rb')
sn=f.read(9)
Tu=struct.unpack('if?', sn)
                       #从字节串 sn 中还原出 1 个整数、1 个浮点数和 1 个布尔值,并返回元组
```

```
print(tu)
n=tu[0]
x=tu[1]
bl=tu[2]
print 'n=', n
print 'x=', x
print 'bl=', bl
s=f.read(9)
f.close()
print 's=', s
```

## 7.4 文件级操作

如果仅需要对文件内容进行读写，可以使用7.1节中介绍的文件对象；如果需要处理文件路径，可以使用os.path模块中的对象和方法；如果需要使用命令行读取文件内容可以使用fileinput模块；创建临时文件和文件夹可以使用tempfile模块；另外，Python 3.4的pathlib模块提供了大量用于表示和处理文件系统路径的类。

### 7.4.1 os与os.path模块

os模块除了提供使用操作系统功能和访问文件系统的简便方法之外，还提供了大量文件级操作的方法，如表7-4所示。os.path模块提供了大量用于路径判断、切分、连接以及文件夹遍历的方法，如表7-5所示。

表7-4 os模块常用文件操作方法

| 方　　法 | 功能说明 |
|---|---|
| access(path,mode) | 按照mode指定的权限访问文件 |
| open(path,flags,mode=0o777,*,dir_fd=None) | 按照mode指定的权限打开文件，默认权限为可读、可写、可执行 |
| chmod(path,mode,*,dir_fd=None,follow_symlinks=True) | 改变文件的访问权限 |
| remove(path) | 删除指定的文件 |
| rename(src,dst) | 重命名文件或目录 |
| stat(path) | 返回文件的所有属性 |
| fstat(path) | 返回打开的文件的所有属性 |
| listdir(path) | 返回path目录下的文件和目录列表 |
| startfile(filepath [,operation]) | 使用关联的应用程序打开指定文件 |

表 7-5　os. path 模块常用文件操作方法

| 方　　法 | 功 能 说 明 |
| --- | --- |
| abspath(path) | 返回绝对路径 |
| dirname(p) | 返回目录的路径 |
| exists(path) | 判断文件是否存在 |
| getatime(filename) | 返回文件的最后访问时间 |
| getctime(filename) | 返回文件的创建时间 |
| getmtime(filename) | 返回文件的最后修改时间 |
| getsize(filename) | 返回文件的大小 |
| isabs(path) | 判断 path 是否为绝对路径 |
| isdir(path) | 判断 path 是否为目录 |
| isfile(path) | 判断 path 是否为文件 |
| join(path, * paths) | 连接两个或多个 path |
| split(path) | 对路径进行分割，以列表形式返回 |
| splitext(path) | 从路径中分割文件的扩展名 |
| splitdrive(path) | 从路径中分割驱动器的名称 |
| walk(top,func,arg) | 遍历目录 |

下面通过几个示例来演示 os 和 os. path 模块的用法。

```
>>> import os
>>> import os.path
>>> os.path.exists('test1.txt')
False
>>> os.rename('C:\\test1.txt', 'D:\\test2.txt')
                                    #此时'C:\\test1.txt'不存在出错信息
>>> os.rename('C:\\dfg.txt', 'D:\\test2.txt')
                                    #os.rename()可以实现文件的改名和移动
>>> os.path.exists('C:\\dfg.txt')
False
>>> os.path.exists('D:\\dfg.txt')
False
>>> os.path.exists('D:\\test2.txt')
True
>>> path='D:\\mypython_exp\\new_test.txt'
>>> os.path.dirname(path)
'D:\\mypython_exp'
>>> os.path.split(path)
('D:\\mypython_exp', 'new_test.txt')
```

```
>>> os.path.splitdrive(path)
('D:', '\\mypython_exp\\new_test.txt')
>>> os.path.splitext(path)
('D:\\mypython_exp\\new_test', '.txt')
```

下面的代码可以列出当前目录下所有扩展名为.pyc的文件，其中用到了列表推导式，可以查阅前面的2.1.9节了解相关知识。

```
>>> import os
>>> print ([fname for fname in os.listdir (os.getcwd ()) if os.path.isfile
(fname) and fname.endswith('.pyc')])
['consts.pyc', 'database_demo.pyc', 'nqueens.pyc']
```

下面的代码用来将当前目录的所有扩展名为.html的文件重命名为扩展名为.htm的文件。

```
import os

file_list=os.listdir(".")
for filename in file_list:
    pos=filename.rindex(".")
    if filename[pos+1:]=="html":
        newname=filename[:pos+1]+"htm"
        os.rename(filename, newname)
        print(filename+"更名为: "+newname)
```

当然，也可以改写为下面的简洁而等价的代码：

```
import os

file_list=[filename for filename in os.listdir(".") if filename.endswith
('.html')]
for filename in file_list:
    newname=filename[:-4]+'htm'
    os.rename(filename, newname)
    print(filename+"更名为: "+newname)
```

### 7.4.2 shutil 模块

shutil模块也提供了大量的方法支持文件和文件夹操作，详细的方法列表可以使用dir(shutil)进行查看。

```
>>> import shutil
>>> dir(shutil)
['Error', 'ExecError', 'ReadError', 'RegistryError', 'SameFileError',
'SpecialFileError', '_ARCHIVE_FORMATS', '_BZ2_SUPPORTED', '_UNPACK_FORMATS',
```

```
'__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_basename', '_call_external_zip',
'_check_unpack_options', '_copyxattr', '_destinsrc', '_ensure_directory',
'_find_unpack_format', '_get_gid', '_get_uid', '_make_tarball', '_make_
zipfile', '_ntuple_diskusage', '_rmtree_safe_fd', '_rmtree_unsafe',
'_samefile', '_unpack_tarfile', '_unpack_zipfile', '_use_fd_functions',
'abspath', 'chown', 'collections', 'copy', 'copy2', 'copyfile', 'copyfileobj',
'copymode', 'copystat', 'copytree', 'disk_usage', 'errno', 'fnmatch', 'get_
archive_formats', 'get_terminal_size', 'get_unpack_formats', 'getgrnam',
'getpwnam', 'ignore_patterns', 'make_archive', 'move', 'nt', 'os', 'register_
archive_format', 'register_unpack_format', 'rmtree', 'stat', 'sys', 'tarfile',
'unpack_archive', 'unregister_archive_format', 'unregister_unpack_format',
'which']
```

例如，下面的代码使用该模块的 copyfile()方法复制文件。

```
>>> import shutil
>>> shutil.copyfile('C:\\dir.txt', 'C:\\dir1.txt')
```

下面的代码将 C:\Python34\Dlls 文件夹以及该文件夹中所有文件压缩至 D:\a. zip 文件。

```
>>> shutil.make_archive('D:\\a', 'zip', 'C:\\Python34', 'Dlls')
'D:\\a.zip'
```

而下面的代码则将刚压缩得到的文件 D:\a. zip 解压缩至 D:\a_unpack 文件夹。

```
>>> shutil.unpack_archive('D:\\a.zip', 'D:\\a_unpack')
```

下面的代码使用 shutil 模块的方法删除刚刚解压缩得到的文件夹。

```
>>> shutil.rmtree('D:\\a_unpack')
```

## 7.5 目录操作

除了支持文件操作，os 和 os. path 模块还提供了大量的目录操作方法，os 模块常用目录操作方法与成员如表 7-6 所示，可以通过 dir(os. path)查看 os. path 模块更多关于目录操作的方法。

**表 7-6 os 模块常用目录操作方法与成员**

| 方法 | 功能说明 |
| --- | --- |
| mkdir(path[,mode=0777]) | 创建目录 |
| makedirs(path1/path2…,mode=511) | 创建多级目录 |
| rmdir(path) | 删除目录 |

续表

| 方　　法 | 功 能 说 明 |
| --- | --- |
| removedirs(path1/path2…) | 删除多级目录 |
| listdir(path) | 返回指定目录下的文件和目录信息 |
| getcwd() | 返回当前工作目录 |
| get_exec_path() | 返回可执行文件的搜索路径 |
| chdir(path) | 把 path 设为当前工作目录 |
| walk(top,topdown=True,onerror=None) | 遍历目录树,该方法返回一个元组,包括 3 个元素:所有路径名、所有目录列表与文件列表 |
| sep | 当前操作系统所使用的路径分隔符 |
| extsep | 当前操作系统所使用的文件扩展名分隔符 |

下面的代码演示了如何使用 os 模块的方法来查看、改变当前工作目录,以及创建与删除目录。

```
>>> import os
>>> os.getcwd()                              #返回当前工作目录
'C:\\Python27'
>>> os.mkdir(os.getcwd()+'\\temp')           #创建目录
>>> os.chdir(os.getcwd()+'\\temp')           #改变当前工作目录
>>> os.getcwd()
'C:\\Python27\\temp'
>>> os.mkdir(os.getcwd()+'\\test')
>>> os.listdir('.')
['test']
>>> os.rmdir('test')                         #删除目录
>>> os.listdir('.')
[]
```

如果需要遍历指定目录下所有子目录和文件,可以使用递归的方法,如下面的代码所示。

```
import os

def visitDir(path):
    if not os.path.isdir(path):
        print('Error:''', path, ''' is not a directory or does not exist.')
        return
    for lists in os.listdir(path):
        sub_path=os.path.join(path, lists)
        print(sub_path)
        if os.path.isdir(sub_path):
```

```
            visitDir(sub_path)

visitDir('E:\\test')
```

下面的代码则使用 os 模块的 walk()方法进行指定目录的遍历。

```
import os

def visitDir2(path):
    if not os.path.isdir(path):
        print('Error:''', path, ''' is not a directory or does not exist.')
        return
    list_dirs=os.walk(path)
    for root, dirs, files in list_dirs:       #遍历该元组的目录和文件信息
        for d in dirs:
            print(os.path.join(root, d))      #获取完整路径
        for f in files:
            print(os.path.join(root, f))      #获取文件绝对路径

visitDir2('h:\\music')
```

也可以使用 os. path 模块的 walk()方法遍历目录,如下面的代码:

```
def visitDir3(arg, dirname, names):
    for filepath in names:
        print(os.path.join(dirname, filepath))

os.path.walk('h:\\music', visitDir3, ())
```

下面的代码可以用来删除当前文件夹以及所有子文件夹中特定类型的文件,其中要删除的文件类型可以在当前文件夹下的配置文件 filetypes. txt 中进行定义,每个文件类型的扩展名占一行。

```
from os.path import isdir, join, splitext
from os import remove, listdir, getcwd

filetypes=[]
def delCertainFiles(directory):
    for filename in listdir(directory):
        temp=join(directory, filename)
        if isdir(temp):
            delCertainFiles(temp)
        elif splitext(temp)[1] in filetypes: #check file extension name
            remove(temp)
            print(temp, ' deleted....')
```

```
def readFileTypes():
    global filetypes
    with open('filetypes.txt', 'r') as fp:
        filetypes=fp.readlines()
    filetypes=[ext.strip() for ext in filetypes]

def main():
    readFileTypes()
    print(filetypes)
    delCertainFiles(getcwd())

main()
```

## 7.6 高级话题

在本章最后，我们一起来看几个高级话题，包括计算文件 MD5 值、判断文件类型、Excel 文件读写等等。

（1）计算 CRC32 值。下面的代码分别使用 zlib 和 binascii 模块的方法来计算任意字符串的 CRC32 值，该代码经过简单修改，即可用来计算文件的 CRC32 值。

```
>>> import zlib
>>> print(zlib.crc32('1234'))
-1679564637
>>> print(zlib.crc32('111'))
1298878781
>>> print(zlib.crc32('SDIBT'))
2095416137
>>> import binascii
>>> binascii.crc32('SDIBT')
2095416137
```

（2）计算文本文件中最长行的长度。

方法一：

```
f=open('D:\\test.txt', 'r')
allLineLens=[len(line.strip()) for line in f]
f.close()
longest=max(allLineLens)
print(longest)
```

方法二：

```
f=open('D:\\test.txt', 'r')
longest=max(len(line.strip()) for line in f)
f.close()
```

```
print(longest)
```

当然,为了实现这个功能,还有很多别的写法,不妨大胆尝试一下,争取写出更加简洁、更加优雅、更加 Pythonic 的代码。

(3) 计算字符串 MD5 值。MD5 值可以用来判断文件发布之后是否被篡改,对于文件完整性保护具有重要意义。

```
>>> import hashlib
>>> import md5
>>> md5value=hashlib.md5()
>>> md5value.update('12345')
>>> md5value=md5value.hexdigest()
>>> print(md5value)
827ccb0eea8a706c4c34a16891f84e7b
>>> md5value=md5.md5()
>>> md5value.update('12345')
>>> md5value=md5value.hexdigest()
>>> print(md5value)
827ccb0eea8a706c4c34a16891f84e7b
```

对上面的代码稍加完善,即可实现自己的 MD5 计算器,例如:

```
import hashlib
import os
import sys

fileName=sys.argv[1]
if os.path.isfile(fileName):
    with open(fileName, 'r') as fp:
        lines=fp.readlines()
    data=''.join(lines)
    print(hashlib.md5(data).hexdigest())
```

将上面的代码保存为文件 CheckMD5OfFile.py,然后计算指定文件的 MD5 值,对该文件进行微小修改后再次计算其 MD5 值。可以发现,哪怕只是修改了一点点内容,MD5 值的变化也是非常大的,如图 7-1 所示。

```
C:\Python27>echo hello world > test.txt

C:\Python27>type test.txt
hello world

C:\Python27>CheckMD5OfFile.py test.txt
edf02967b5ae298061db1f1ad2a6b6b4

C:\Python27>
C:\Python27>echo hello world1 > test.txt

C:\Python27>CheckMD5OfFile.py test.txt
04ddb543e0cf5585aa6d72420cfe2e95

C:\Python27>type test.txt
hello world1
```

图 7-1 计算文件 MD5 值

另外,也可以使用 ssdeep 工具来计算文件的模糊哈希值或分段哈希值,或者编写 Python 程序调用 ssdeep 提供的 API 函数来计算文件的模糊哈希值,模糊哈希值可以用来比较两个文件的相似百分比。

```
>>> from ssdeep import ssdeep
>>> s=ssdeep()
>>> print s.hash_file(filename)
```

对于某些恶意软件来说，可能会对自身进行加壳或加密，真正运行时再进行脱壳或解密，这样一来，会使得磁盘文件的哈希值和内存中脱壳或解密后进程的哈希值相差很大。因此，根据磁盘文件和其相应的进程之间模糊哈希值的相似度可以判断该文件是否包含自修改代码，并以此来判断其为恶意软件的可能性。

(4) 判断一个文件是否为 GIF 图像文件。任何一种文件都具有专门的文件头结构，在文件头中存放了大量的信息，其中就包括该文件的类型。通过文件头信息来判断文件类型的方法可以得到更加准确的信息，而不依赖于文件扩展名。

```
>>> def is_gif(fname):
    f=open(fname, 'r')
    first4=tuple(f.read(4))
    print first4
    f.close()
    return first4==('G', 'I', 'F', '8')
>>> is_gif('C:\\test.gif')
('G', 'I', 'F', '8')
True
>>> is_gif('C:\\dir.txt')
False
```

(5) 比较两个文本文件是否相同。这里使用到了 difflib 模块的 SequenceMatcher() 方法，检测结果相对还算清晰，请大家运行下面的代码并查看结果。

```
import difflib
A=file('C:\\dir.txt', 'r')
B=file('C:\\dir1.txt', 'r')
contextA=A.read()
contextB=B.read()
s=difflib.SequenceMatcher(lambda x: x=="", contextA, contextB)
result=s.get_opcodes()
for tag, i1, i2, j1, j2 in result:
    print("%s contextA[%d:%d]=%s contextB[%d:%d]=%s"%\
          (tag, i1, i2, contextA [i1:i2], j1, j2, contextB[j1:j2]))
```

(6) 使用 xlwt 模块写入 Excel 文件。xlwt 模块默认没有安装，可以使用 pip 进行安装。

```
from xlwt import *

book=Workbook()
sheet1=book.add_sheet("First")
al=Alignment()
al.horz=Alignment.HORZ_CENTER            #对齐方式
al.vert=Alignment.VERT_CENTER
borders=Borders()
```

```
borders.bottom=Borders.THICK                                #边框样式
style=XFStyle()
style.alignment=al
style.borders=borders
row0=sheet1.row(0)
row0.write(0, 'test', style=style)
book.save(r'D:\test.xls')
```

(7) 使用 xlrd 模块读取 Excel 文件，与 xlwt 模块一样，xlrd 模块也需要单独安装。

```
>>> import xlrd
>>> book=xlrd.open_workbook(r'D:\test.xls')
>>> sheet1=book.sheet_by_name('First')
>>> row0=sheet1.row(0)
>>> print row0[0]
text:u'test'
>>> print row0[0].value
test
```

(8) 使用 Pywin32 操作 Excel 文件。Pywin32 模块需要单独安装，这是一个功能非常强大的模块，提供了 Windows 底层 API 函数的封装，使得可以在 Python 中直接调用 Windows API 函数，支持大量的 Windows 底层操作，后面章节还会用到该模块的其他功能。

```
xlApp=win32com.client.Dispatch('Excel.Application')        #打开 Excel
xlBook=xlApp.Workbooks.Open('D:\\1.xls')
xlSht=xlBook.Worksheets('sheet1')
aaa=xlSht.Cells(1, 2).Value
xlSht.Cells(2, 3).Value=aaa
xlBook.Close(SaveChanges=1)
del xlApp
```

(9) 检查 Word 文档的连续重复字。在 Word 文档中，经常会由于键盘操作不小心而使得文档中出现连续的重复字，例如“用户的的资料”或“需要需要用户输入”之类的情况。下面的代码使用 Pywin32 模块中的 win32com 对 Word 文档进行检查并提示类似的重复汉字或标点符号。

```
import sys
from win32com import client

#filename=sys.argv[1]
filename=r'c:\test.doc'
word=client.Dispatch('Word.Application')
#newdoc=word.Documents.Add()
doc=word.Documents.Open(filename)
content=str(doc.Content)
```

```
doc.Close()
#newdoc.Close()
word.Quit()

repeatedWords=[]

lens=len(content)
for i in range(lens-2):
    ch=content[i]
    ch1=content[i+1]
    ch2=content[i+2]
    if (u'\u4e00'<=ch<=u'\u9fa5' or ch in (',', '。', '、')):
        if ch==ch1 and ch+ch1 not in repeatedWords:
            print(ch+ch1)
            repeatedWords.append(ch+ch1)
        elif ch==ch2 and ch+ch1+ch2 not in repeatedWords:
            print(ch+ch1+ch2)
            repeatedWords.append(ch+ch1+ch2)
```

## 本章小结

（1）文件操作在各类软件开发中均占有重要的地位。

（2）二进制文件无法直接读取和理解其内容，必须了解其文件结构和所使用的序列化规则并使用正确的反序列化方法。

（3）Python 内置了文件对象，通过 open()函数即可以指定模式打开指定文件并创建文件对象。

（4）Python 中常用的序列化模块有 struct、pickle、json、marshal 和 shelve，其中 pickle 有 C 语言实现的 cPickle，速度约提高 1000 倍，应优先考虑使用。

（5）文件对象的读、写方法都会自动改变文件指针位置。

（6）Python 2.x 和 Python 3.x 对文件对象的读、写方法解释略有不同，尤其对于读写内容包含中文的情况。

（7）os、os.path 和 shutil 模块提供了大量用于文件和文件夹操作的方法，包括文件和文件夹的移动、复制、删除、重命名以及压缩与解压缩等等。

## 习题

7.1　假设有一个英文文本文件，编写程序读取其内容，并将其中的大写字母变为小写字母，小写字母变为大写字母。

7.2　编写程序，将包含学生成绩的字典保存为二进制文件，然后再读取内容并显示。

7.3　使用 shutil 模块中的 move()方法进行文件移动。

7.4 简单解释文本文件与二进制文件的区别。

7.5 编写代码，将当前工作目录修改为“C:\”，并验证，最后将当前工作目录恢复为原来的目录。

7.6 编写程序，用户输入一个目录和一个文件名，搜索该目录及其子目录中是否存在该文件。

7.7 文件对象的________方法用来把缓冲区的内容写入文件，但不关闭文件。

7.8 os.path 模块中的________方法用来测试指定的路径是否为文件。

7.9 os 模块的________方法用来返回包含指定文件夹中所有文件和子文件夹的列表。

# 第 8 章　异常处理结构与程序调试

几乎每种高级编程语言都提供了异常处理结构,Python 也不例外。简单地说,异常是指程序运行时引发的错误,引发错误的原因有很多,例如除零、下标越界、文件不存在、网络异常、类型错误、名字错误、字典键错误、磁盘空间不足,等等。如果这些错误得不到正确的处理将会导致程序终止运行,而合理地使用异常处理结构可以使得程序更加健壮,具有更强的容错性,不会因为用户不小心的错误输入或其他运行时原因而造成程序终止。或者,也可以使用异常处理结构为用户提供更加友好的提示。另外,程序出现异常或错误之后是否能够调试程序并快速定位和解决存在的问题也是程序员综合水平和能力的重要体现方式之一。本章首先介绍 Python 异常以及异常处理结构,然后介绍几种不同的 Python 程序调试技术。

## 8.1　基本概念

什么是异常呢？让我们先来看几个示例。

```
>>> x, y=10, 5
>>> a=x / y
>>> print(A)            #拼写错误,Python 区分变量名等标识符字母的大小写
Traceback (most recent call last):
File "<pyshell#2>", line 1, in<module>
print A
NameError: name 'A' is not defined
>>> 10 * (1/0)          #除 0 错误
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4+spam*3            #使用了未定义的变量,与拼写错误的情形相似
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' +2              #对象类型不支持特定的操作
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

在前面的章节中,你肯定已经注意到类似的信息,没错,这就是 Python 异常的标准表现形式。熟练运用异常处理机制对于提高程序的健壮性和容错性具有重要的作用,同时也可以把 Python 晦涩难懂的错误提示转换为友好的提示显示给最终用户。

异常处理是指因为程序执行过程中出错而在正常控制流之外采取的行为。严格来说，语法错误和逻辑错误不属于异常，但有些语法或逻辑错误往往会导致异常，例如，由于大小写拼写错误而试图访问不存在的对象，或者试图访问不存在的文件，等等。当 Python 检测到一个错误时，解释器就会指出当前程序流已无法继续执行下去，这时候就出现了异常。当程序执行过程中出现错误时 Python 会自动引发异常，程序员也可以通过 raise 语句显式地引发异常。

需要注意的是，尽管异常处理机制非常重要也非常有效，但是不建议使用异常来代替常规的检查，例如必要的 if...else 判断。在编程时应避免过多依赖于异常处理机制来提高程序健壮性。

## 8.2 Python 异常类与自定义异常

下面较为完整地展示了 Python 内建异常类的继承层次。

```
BaseException
+--SystemExit
+--KeyboardInterrupt
+--GeneratorExit
+--Exception
      +--StopIteration
      +--ArithmeticError
      |     +--FloatingPointError
      |     +--OverflowError
      |     +--ZeroDivisionError
      +--AssertionError
      +--AttributeError
      +--BufferError
      +--EOFError
      +--ImportError
      +--LookupError
      |     +--IndexError
      |     +--KeyError
      +--MemoryError
      +--NameError
      |     +--UnboundLocalError
      +--OSError
      |     +--BlockingIOError
      |     +--ChildProcessError
      |     +--ConnectionError
```

```
 |       |       +--BrokenPipeError
 |       |       +--ConnectionAbortedError
 |       |       +--ConnectionRefusedError
 |       |       +--ConnectionResetError
 |       +--FileExistsError
 |       +--FileNotFoundError
 |       +--InterruptedError
 |       +--IsADirectoryError
 |       +--NotADirectoryError
 |       +--PermissionError
 |       +--ProcessLookupError
 |       +--TimeoutError
 +--ReferenceError
 +--RuntimeError
 |       +--NotImplementedError
 +--SyntaxError
 |       +--IndentationError
 |               +--TabError
 +--SystemError
 +--TypeError
 +--ValueError
 |       +--UnicodeError
 |               +--UnicodeDecodeError
 |               +--UnicodeEncodeError
 |               +--UnicodeTranslateError
 +--Warning
      +--DeprecationWarning
      +--PendingDeprecationWarning
      +--RuntimeWarning
      +--SyntaxWarning
      +--UserWarning
      +--FutureWarning
      +--ImportWarning
      +--UnicodeWarning
      +--BytesWarning
      +--ResourceWarning
```

如果需要，可以继承 Python 内置异常类来实现自定义的异常类，例如：

```
class ShortInputException(Exception):
```

```
    '''自定义异常类。'''
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length=length
        self.atleast=atleast
try:
    s=raw_input('请输入 -->')
    if len(s)<3:
        raise ShortInputException(len(s), 3)
except EOFError:
    print('您输入了一个结束标记 EOF')
except ShortInputException, x:
    print('ShortInputException: 输入的长度是 %d, 长度至少应是 %d' %(x.length, x.
atleast))
else:
    print('没有异常发生.')
```

再例如下面的示例:

```
>>> class MyError(Exception):
        def __init__(self, value):
            self.value=value
        def __str__(self):
            return repr(self.value)
>>> try:
        raise MyError(2 * 2)
    except MyError as e:
        print('My exception occurred, value:', e.value)
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

如果自己编写的某个模块需要抛出多个不同但相关的异常,可以先创建一个基类,然后创建多个派生类分别表示不同的异常。

```
class Error(Exception):          #创建基类
    pass

class InputError(Error):         #派生类 InputError
    """Exception raised for errors in the input.
    Attributes:
        expression --input expression in which the error occurred   message --
explanation of the error
    """
```

```
        def __init__(self, expression, message):
            self.expression=expression
            self.message=message

    class TransitionError(Error):          #派生类 TransitionError
        """Raised when an operation attempts a state transition that's not allowed.
        Attributes:
            previous --state at beginning of transition
            next --attempted new state
            message --explanation of why the specific transition is not allowed
        """
        def __init__(self, previous, next, message):
            self.previous=previous
            self.next=next
            self.message=message
```

## 8.3 Python 中的异常处理结构

### 8.3.1 try...except 结构

异常处理结构中最常见也最基本的是 try...except...结构。其中 try 子句中的代码块包含可能出现异常的语句，而 except 子句用来捕捉相应的异常，except 子句中的代码块，则用来处理异常。如果 try 中的代码块没有出现异常，则继续往下执行异常处理结构后面的代码；如果出现异常并且被 except 子句捕获，则执行 except 子句中的异常处理代码；如果出现异常但没有被 except 捕获，则继续往外层抛出；如果所有层都没有捕获并处理该异常，则程序终止并将该异常抛给最终用户。该结构语法如下：

```
try:
    try 块                           #被监控的语句，可能会引发异常
except Exception[, reason]:
    except 块                        #处理异常的代码
```

如果需要捕获所有类型的异常，可以使用 BaseException，即 Python 异常类的基类，代码格式如下：

```
try:
    …
except BaseException, e:
    except 块                        #处理所有错误
```

上面的结构可以捕获所有异常，尽管这样做很安全，但是一般并不建议这样做。对于异常处理结构，一般的建议是尽量显式捕捉可能会出现的异常，并且有针对性地编写代码进行处理，因为在实际应用开发中，很难使用同一段代码去处理所有类型的异常。当然，

为了避免遗漏没有得到处理的异常干扰程序的正常执行，在捕捉了所有可能想到的异常之后，也可以使用异常处理结构的最后一个 except 来捕捉 BaseException。

下面的代码演示了 try...except...结构的用法，代码运行后提示用户输入内容，如果输入的是数字，则循环结束，否则一直提示用户输入正确格式的内容。

```
>>> while True:
        try:
            x=int(input("Please enter a number: "))
            break
        except ValueError:
            print("That was not a valid number.  Try again...")
```

在使用时，except 子句可以在异常类名字后面指定一个变量，用来捕获异常的参数或更详细的信息。

```
>>> try:
        raise Exception('spam', 'eggs')
    except Exception as inst:
        print(type(inst))      #the exception instance
        print(inst.args)       #arguments stored in .args
        print(inst)            #__str__ allows args to be printed directly,
                               #but may be overridden in exception subclasses
        x, y=inst.args         #unpack args
        print('x=', x)
        print('y=', y)

<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x=spam
y=eggs
```

### 8.3.2 try...except...else 结构

另外一种常用的异常处理结构是 try...except...else...语句。正如前面章节中已经提到过，带 else 子句的异常处理结构也是一种特殊形式的选择结构。如果 try 中的代码抛出了异常，并且被某个 except 捕捉，则执行相应的异常处理代码，这种情况下不会执行 else 中的代码；如果 try 中的代码没有抛出任何异常，则执行 else 块中的代码。例如下面的代码：

```
a_list=['China', 'America', 'England', 'France']
print '请输入字符串的序号'
while True:
```

```
    n=input()
    try:
        print a_list[n]
    except IndexError:
        print '列表元素的下标越界或格式不正确,请重新输入字符串的序号'
    else:
        break #结束循环
```

### 8.3.3 带有多个 except 的 try 结构

在实际开发中,同一段代码可能会抛出多个异常,需要针对不同的异常类型进行相应的处理。为了支持多个异常的捕捉和处理,Python 提供了带有多个 except 的异常处理结构,这类似于多分支选择结构。一旦某个 except 捕获了异常,则后面剩余的 except 子句将不会再执行。该结构的语法为:

```
try:
    try 块                      #被监控的语句
except Exception1:
    except 块 1                 #处理异常 1 的语句
except Exception2:
    except 块 2                 #处理异常 2 的语句
```

下面的代码演示了该结构的用法:

```
try:
    x=input('请输入被除数: ')
    y=input('请输入除数: ')
    z=float(x) / y
except ZeroDivisionError:
    print '除数不能为零'
except TypeError:
    print '被除数和除数应为数值类型'
except NameError:
    print '变量不存在'
else:
    print x, '/', y, '=', z
```

将要捕获的异常写在一个元组中,可以使用一个 except 语句捕获多个异常,并且共用同一段异常处理代码,当然,除非确定要捕获的多个异常可以使用同一段代码来处理,否则并不建议这样做。

```
import sys
try:
    f=open('myfile.txt')
```

```
    s=f.readline()
    i=int(s.strip())
except (OSError, ValueError, RuntimeError, NameError):
    pass
```

### 8.3.4 try…except…finally 结构

最后一种常用的异常处理结构是 try…except…finally…结构。在该结构中，finally 子句中的语句块无论是否发生异常都会执行，常用来做一些清理工作以释放 try 子句中申请的资源。语法如下：

```
try:
    …
finally:
    …          #无论如何都会执行的代码
```

例如下面的代码，无论是否发生异常，语句 print(5)都会被执行。

```
>>> try:
    3/0
except:
    print(3)
finally:
    print(5)
3
5
```

再例如下面的代码，无论读取文件是否发生异常，总是能够保证正常关闭该文件。

```
try:
    f=open('test.txt', 'r')
    line=f.readline()
    print line
finally:
    f.close()
```

需要注意的一个问题是，如果 try 子句中的异常没有被捕捉和处理，或者 except 子句或 else 子句中的代码出现了异常，那么这些异常将会在 finally 子句执行完后再次抛出。例如上面的代码，使用异常处理结构的本意是为了防止文件读取操作出现异常而导致文件不能正常关闭，但是如果因为文件不存在而导致文件对象创建失败，那么 finally 子句中关闭文件对象的代码将会抛出异常从而导致程序终止运行。

```
>>> try:
    f=open('test.txt', 'r')
    line=f.readline()
```

```
    print(line)
finally:
    f.close()
Traceback (most recent call last):
  File "<pyshell#17>", line 6, in<module>
    f.close()
NameError: name 'f' is not defined
```

再例如下面的代码，在 try 中的语句出现了异常但是没有得到处理，因此，finally 中的语句执行完以后再次抛出该异常。

```
>>> try:
    3/0
finally:
    print(5)
5
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in<module>
    try:3/0
finally:
    print(5)
ZeroDivisionError: division by zero
```

下面的代码较为完整地演示了这种情况。

```
>>> def divide(x, y):
        try:
            result=x / y
        except ZeroDivisionError:
            print("division by zero!")
        else:
            print("result is", result)
        finally:
            print("executing finally clause")
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

最后，使用带有 finally 子句的异常处理结构时，应尽量避免在 finally 子句中使用 return 语句，否则可能会出现出乎意料的错误，例如下面的代码：

```
>>> def demo_div(a, b):
    try:
        return a/b
    except:
        pass
    finally:
        return -1
>>> demo_div(1, 0)
-1
>>> demo_div(1, 2)
-1
>>> demo_div(10, 2)
-1
```

通过本节的介绍，相信你已经了解和掌握了 Python 异常处理结构的原理和用法。简单总结一下，可以理解为“请求原谅比请求允许要容易”。也就是说，有些代码执行可能会出现错误，也可能不会出现错误，这主要由运行时的各种客观因素决定，此时建议使用异常处理结构。如果使用大量的选择结构来提前判断，仅当满足相应条件时才执行该代码，这些条件判断可能会严重干扰正常的业务逻辑，也会严重降低代码的可读性。

## 8.4 断言与上下文管理

断言与上下文管理可以说是两种比较特殊的异常处理方式，在形式上比异常处理结构要简单一些，能够满足简单的异常处理或条件确认，并且可以与标准的异常处理结构结合使用。

### 8.4.1 断言

断言语句的语法是：

```
assert expression[, reason]
```

当判断表达式 expression 为真时，什么都不做；如果表达式为假，则抛出异常。

assert 语句一般用于对程序某个时刻必须满足的条件进行验证，仅当“__ debug __”为 True 时有效。当 Python 脚本以-O 选项编译为字节码文件时，assert 语句将被移除以提高运行速度。

断言和异常处理结构经常结合使用，例如：

```
>>> try:
    assert 1==2, "1 is not equal 2!"
```

```
except AssertionError, reason:
    print "%s:%s"%(reason.__class__.__name__, reason)
AssertionError:1 is not equal 2!
```

### 8.4.2 上下文管理

使用上下文管理语句 with 可以自动管理资源，在代码块执行完毕后自动还原进入该代码块之前的现场或上下文。不论何种原因跳出 with 块，也不论是否发生异常，总能保证资源被正确释放，大大简化了程序员的工作，常用于文件操作、网络通信之类的场合。

with 语句的语法如下：

```
with context_expr [as var]:
    with 块
```

例如，下面的代码演示了文件操作时 with 语句的用法，使用这样的写法程序员丝毫不用担心忘记关闭文件，当文件处理完以后，将会自动关闭。

```
with open('D:\\test.txt') as f:
    for line in f:
        print(line)
```

## 8.5 用 sys 模块回溯最后的异常

当发生异常时，Python 会回溯异常，给出大量的提示，可能会给程序员的定位和纠错带来一定的困难，这时可以使用 sys 模块来回溯最近一次异常。语法为：

```
import sys
try:
    block
except:
    t=sys.exc_info()
    print(t)
```

sys.exc_info()的返回值是一个三元组(type, value/message, traceback)。其中，type 表示异常的类型，value/message 表示异常的信息或者参数，而 traceback 则包含调用栈信息的对象。

例如，下面的代码演示了其用法：

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in<module>
    1/0
ZeroDivisionError: integer division or modulo by zero>>> import sys
```

```
>>> try:
    1/0
except:
    r=sys.exc_info()
    print(r)
(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero',),
<traceback object at 0x000000000375C788>)
```

下面的代码演示了标准的异常跟踪和 sys. exc_info()之间的区别,可以看出,sys. exc_info()可以直接定位最终引发异常的原因,结果也比较简洁,但是缺点是难以直接确定引发异常的代码位置:

```
>>> def A():
    1/0
>>> def B():
    A()
>>> def C():
    B()
>>> C()
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in<module>
    C()
  File "<pyshell#34>", line 2, in C
    B()
  File "<pyshell#31>", line 2, in B
    A()
  File "<pyshell#28>", line 2, in A
    1/0
ZeroDivisionError: integer division or modulo by zero
>>> try:
    C()
except:
    r=sys.exc_info()
    print(r)
(<type 'exceptions.ZeroDivisionError'>, ZeroDivisionError('integer
division or modulo by zero',),<traceback object at 0x0134C990>)
```

## 8.6 使用 IDLE 调试代码

当程序运行发生错误或者得到了非预期的结果时,是否能够熟练地对程序进行调试并快速定位和解决问题是体现程序员综合能力的重要标准之一。

几乎任何一种集成开发环境都提供了代码调试功能,Python 标准开发环境 IDLE 也不例外。使用 IDLE 的调试功能时,首先单击 IDLE 的 Debug→Debugger 菜单命令打开

调试器窗口，然后打开并运行要调试的程序，最后切换到调试器窗口使用其中的控制按钮进行调试。如图 8-1 所示为 IDLE 调试窗口及其功能简要介绍，可以使用调试按钮对程序进行单步执行，实时查看变量的当前值并跟踪其变化过程，对于理解程序内部工作原理和发现程序中存在的问题非常有帮助。图 8-2 和图 8-3 是 7.2 节例 7-5（读取文本文件中的数值并排序后写入另一个文本文件）程序调试过程的两个截图，从中可以看到列表 data 中的数据变化过程。另外，从调试过程中也可以很容易地看出，列表推导式的内部执行原理仍是循环，只是形式上比较简单。

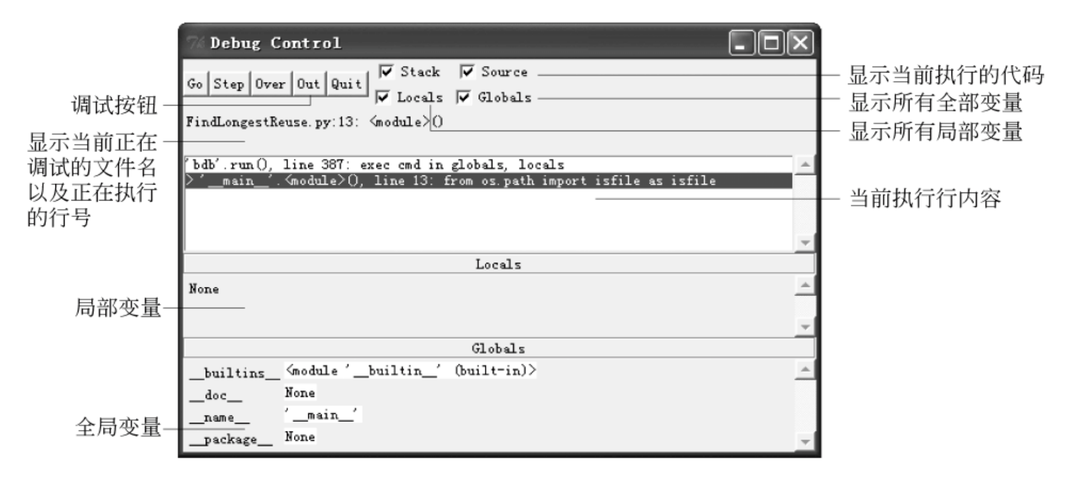



图 8-1　IDLE 调试器窗口

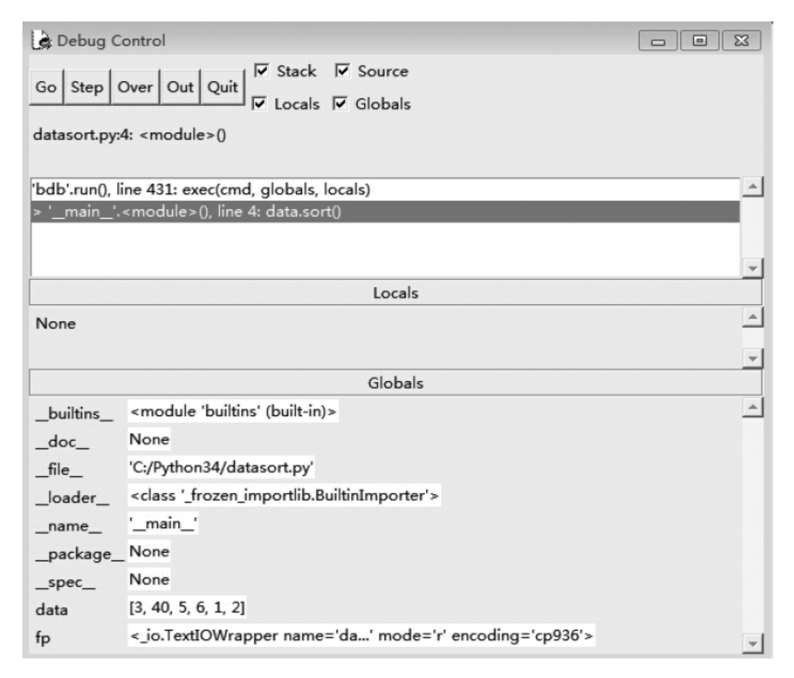



图 8-2　程序调试截图（一）

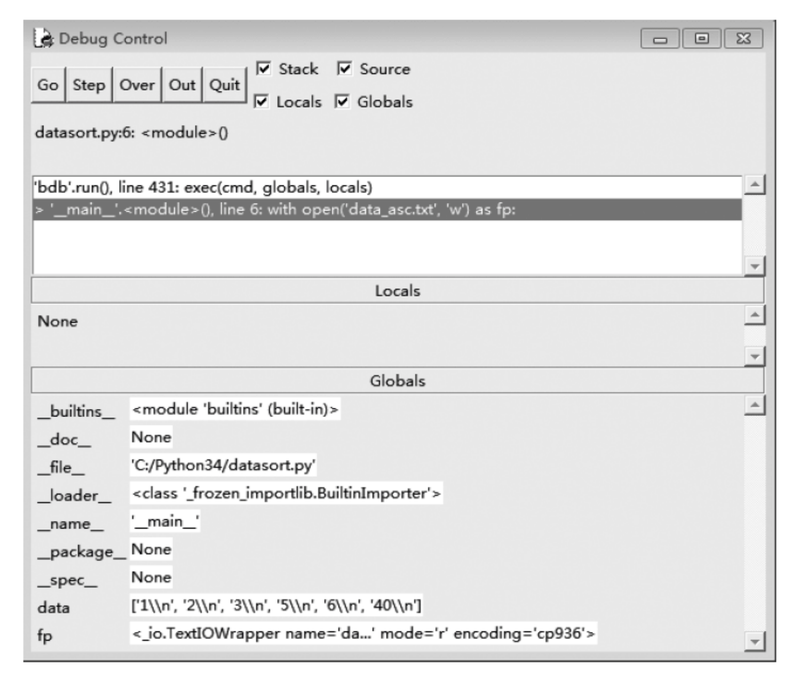



图 8-3　程序调试截图(二)

# 8.7　使用 pdb 模块调试程序

## 8.7.1　pdb 模块常用命令

pdb 是 Python 自带的交互式源代码调试模块，源代码文件为 pdb.py，感兴趣的读者可以在 Python 安装目录下找到该文件进行阅读并理解其工作原理。pdb 模块需要导入后才能使用其中的功能，使用该模块可以完成代码调试的绝大部分功能，包括设置/清除(条件)断点、启用/禁用断点、单步执行、查看栈帧、查看变量值、查看当前执行位置、列出源代码、执行任意 Python 代码或表达式等。pdb 还支持事后调试，可在程序控制下被调用，并且可以通过 pdb 和 cmd 接口对该调试器进行扩展。pdb 模块常用调试命令如表 8-1 所示。

**表 8-1　常用 pdb 调试命令**

| 简写/完整命令 | 用法示例 | 解　释 |
|---|---|---|
| a(rgs) | | 显示当前函数中的参数 |
| b(reak) [[filename:]lineno \| function[, condition]] | b 173 | 在 173 行设置断点 |
| | b function | 在 function 函数第一条可执行语句位置设置断点 |
| | b | 不带参数则列出所有断点，包括每个断点的触发次数、当前忽略计数以及与之关联的条件 |
| | b 175, condition | 设置条件断点，仅当 condition 的值为 True 时该断点有效 |

续表

| 简写/完整命令 | 用法示例 | 解释 |
|---|---|---|
| cl(ear) [filename:lineno \| bpnumber [bpnumber ...]] | cl | 清除所有断点 |
| | cl filename:lineno | 删除指定文件指定行的所有断点 |
| | cl 3 5 9 | 删除第3、5、9个断点 |
| condition bpnumber [condition] | condition 3 a<b | 仅当a<b时3号断点有效 |
| | condition 3 | 将3号断点设置为无条件断点 |
| continue | | 继续运行至下一个断点或脚本结束 |
| disable [bpnumber [bpnumber ...]] | disable 3 5 | 禁用第3、5个断点，禁用后断点仍存在，可以再次被启用 |
| d(own) | | 在栈跟踪器中向下移动一个栈帧 |
| enable [bpnumber [bpnumber ...]] | enable n | 启用第n个断点 |
| h(elp) [command] | | 查看pdb帮助 |
| ignore bpnumber [count] | | 为断点设置忽略计数，count默认值为0。若某断点的忽略计数不为0，则每次触发时自动减1，当忽略计数为0时该断点处于活动状态 |
| j(ump) | j 20 | 跳至第20行继续运行 |
| l(ist) [first [, last]] | l | 列出脚本清单，默认11行 |
| | l m, n | 列出从第m行到第n之间的脚本代码 |
| | l m | 列出从第m行开始的11行代码 |
| n(ext) | | 执行下一条语句，遇到函数时不进入其内部 |
| p(rint) | p i | 打印变量i的值 |
| q(uit) | | 退出pdb调试环境 |
| r(eturn) | | 一直运行至当前函数返回 |
| tbreak | | 设置临时断点，该类型断点只被中断一次，触发后该断点自动删除 |
| step | | 执行下一条语句，遇到函数时进入其内部 |
| u(p) | | 在栈跟踪器中向上移动一个栈帧 |
| w(here) | | 查看当前栈帧 |
| [!]statement | | 在pdb中执行语句，!与要执行的语句之间不需要空格，任何非pdb命令都被解释为Python语句并执行，甚至可以调用函数或修改当前上下文中变量的值 |
| | | 直接回车则默认执行上一个命令，但如果上一个命令是list，则会列出接下来的11行代码 |

### 8.7.2 使用 pdb 模块调试 Python 程序

可以通过三种不同的形式来使用 pdb 模块提供的调试功能，分别为在交互模式下调试特定的代码块、在程序中显式插入断点以及把 pdb 作为模块来调试程序，接下来分别进行简要介绍。

(1) 在交互模式下使用 pdb 模块提供的功能可以直接调试语句块、表达式、函数等多种脚本，常用的调试方法有：

- pdb.run(statement[, globals[, locals]])——调试指定语句，可选参数 globals 和 locals 用来指定代码执行的环境，默认是“__ main __”模块的字典。
- pdb.runeval(expression[, globals[, locals]])——返回表达式的值，可选参数 globals 和 locals 的含义与上面的 run()函数一样。
- pdb.runcall(function[, argument, ...])——调试指定函数。
- pdb.post_mortem([traceback])——进入指定 traceback 对象的事后调试模式，如果没有指定 traceback 对象，则使用当前正在处理的一个异常。

例如，下面的代码演示了如何调试一个函数，其中“(Pdb)”为提示符，在后面输入并执行前面表 8-1 中介绍的命令即可。

```
>>> import pdb
>>> def f():
    x=5
    print x
>>> pdb.runcall(f)
><pyshell#5>(2)f()
(Pdb) n
><pyshell#5>(3)f()
(Pdb) l
[EOF]
(Pdb) p x
5
(Pdb) n
5
--Return--
><pyshell#5>(3)f()->None
(Pdb) n
>>>
```

(2) 在程序中嵌入断点来实现调试功能。

在程序中首先导入 pdb 模块，然后使用 pdb.set_trace()在需要的位置设置断点。如

果程序中存在通过该方法调用显式插入的断点，那么在命令提示符环境下执行该程序或双击执行程序时将自动打开 pdb 调试环境，即使该程序当前不处于调试状态。例如，下面的程序：

```
IsPrime.py:
import pdb

n=37
pdb.set_trace()
for i in range(2, n):
    if n%i==0:
        print 'No'
        break
else:
    print 'Yes'
```

由于使用 pdb 设置的断点，运行后自动打开调试模式，如图 8-4 所示。

```
>>> ================================ RESTART =
>>>
> c:\python27\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) n
> c:\python27\isprime.py(6)<module>()
-> if n%i==0:
(Pdb) n
> c:\python27\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) p i
2
(Pdb) n
> c:\python27\isprime.py(6)<module>()
-> if n%i==0:
(Pdb) n
> c:\python27\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) p i
3
(Pdb)
```

图 8-4 自动打开调试模式(一)

在命令提示符环境中运行该程序同样自动打开调试模式，如图 8-5 所示。

(3) 使用命令行调试程序。

在命令行提示符下执行“python -m pdb 脚本文件名”，则直接进入调试环境；当调试结束或程序正常结束以后，pdb 将重启该程序。例如，把上面的程序 IsPrime. py 中 pdb 模块的导入和断点插入函数都删除，然后在命令提示符环境中使用调试模式运行，如图 8-6 所示。

```
C:\WINDOWS\system32\cmd.exe - IsPrime.py
C:\Documents and Settings\Administrator>cd \python27

C:\Python27>IsPrime.py
> c:\python27\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) p i
*** NameError: NameError("name 'i' is not defined",)
(Pdb) n
> c:\python27\isprime.py(6)<module>()
-> if n%i==0:
(Pdb) p i
2
(Pdb) l
  1     import pdb
  2
  3     n=37
  4     pdb.set_trace()
  5     for i in range(2,n):
  6  ->     if n%i==0:
  7             print 'No'
  8             break
  9     else:
 10         print 'Yes'
[EOF]
(Pdb)
```

图 8-5　自动打开调试模式(二)

```
C:\Python27>IsPrime.py
Yes

C:\Python27>python -m pdb IsPrime.py
> c:\python27\isprime.py(1)<module>()
-> n=37
(Pdb) n
> c:\python27\isprime.py(2)<module>()
-> for i in range(2,n):
(Pdb) w
  c:\python27\lib\bdb.py(400)run()
-> exec cmd in globals, locals
  <string>(1)<module>()
> c:\python27\isprime.py(2)<module>()
-> for i in range(2,n):
(Pdb) n
> c:\python27\isprime.py(3)<module>()
-> if n%i==0:
(Pdb) p i
2
(Pdb) p n
37
(Pdb)
```

图 8-6　以调试模式运行程序

## 本章小结

(1) 程序出现异常或错误后是否能够调试程序并快速定位和解决存在的问题是程序员综合水平和能力的重要体现方式之一。

(2) 异常处理结构可以提高程序的容错性和健壮性,但不建议过多依赖异常处理结构。

(3) 可以继承 Python 内建异常类来实现自定义的异常类。

(4) 可以使用 BaseException 来捕获所有异常,但不建议这样做。

（5）异常处理结构中主要的关键字有 try、except、finally 和 else。

（6）异常处理结构中也可以使用 else 子句，当没有异常发生时执行 else 子句中的代码块。

（7）断言语句 assert 一般用于对程序某个时刻必须满足的条件进行验证。

（8）上下文管理语句 with 在代码块执行完毕后能够自动还原进入代码块之前的现场或上下文，不论是否发生异常总能保证资源被正确释放。

（9）异常处理结构的 finally 子句中的代码仍可能会抛出异常。

（10）可以使用 sys 模块来回溯引发异常的最直接原因。

（11）可以通过三种方式使用 pdb 模块的调试功能：在交互模式下使用 pdb 模块的方法调试指定函数或语句、在程序中显式插入断点、执行 Python 程序时指定 pdb 调试模块。

## 习题

8.1　Python 异常处理结构有哪几种形式？

8.2　异常和错误有什么区别？

8.3　使用 pdb 模块进行 Python 程序调试主要有哪几种用法？

8.4　Python 内建异常类的基类是________________。

8.5　断言语句的语法为____________________。

8.6　Python 上下文管理语句为________________。

# 第 9 章　GUI 编程

在前面所有章节中，我们都是使用控制台应用程序的编写来演示 Python 语言的精妙和强大。然而，对大多数最终用户而言，可能更习惯使用 GUI 程序，毕竟现在已经很少有用户使用控制台的命令行工作模式了，他们更希望使用带有窗口、按钮、菜单、组合框、列表框等 GUI 元素的应用程序。为此，本章介绍 Python 的 GUI 编程，让程序更容易使用和操作。

目前，常用 GUI 工具除了 Python 标准 GUI 库 Tkinter，还有功能强大的跨平台库 wxPython、基于 Java 的 Jython、支持. NET 应用程序的 IronPython 等等。本章以 wxPython 为例来介绍 Python 的 GUI 应用开发，有了 wxPython 的基础，就可以很快掌握和使用其他 GUI 库。

在本书编写时，wxPython 的最新版本是 3. 0. 2. 0，可以登录 wxPython 官方网址 http://wxpython. org/download. php 进行下载并安装。

使用 wxPython 创建 GUI 程序的三个主要步骤为：

(1) 导入 wxPython 包。

(2) 建立框架类：框架类的父类为 wx. Frame，在框架类的构造函数中调用父类的构造函数进行初始化；然后为 frame 类添加各种控件以及事件处理方法，如需在窗体上增加其他控件，可在构造函数中增加代码，如需处理相应事件，可增加框架类的成员函数，并将事件处理方法与相应的控件进行绑定。

(3) 建立主程序：通常需要做 4 件事，即创建应用程序对象、创建框架类对象、显示框架、开始事件循环。执行 frame. Show(True)后，框架才能看得见，执行 app. MainLoop()后，框架才能处理事件。

## 9.1　Frame

Frame 也称为框架或窗体，是所有框架的父类，也是包含标题栏、菜单、按钮等其他控件的容器，运行之后可移动、缩放。

创建 GUI 程序框架时，需要使用继承 wx. Frame 派生出子类，在派生类中调用基类构造函数进行必要的初始化，其构造函数格式为：

```
__init__(self, Window parent, int id=-1, String title=EmptyString, Point pos
=DefaultPosition, Size size= DefaultSize, long style= DEFAULT_FRAME_STYLE,
String name=FrameNameStr)
```

各参数具体含义为：

- parent——框架的父窗体。该值为 None 时表示创建顶级窗体。

- id——新窗体的 wxPython ID 号。可以明确地传递一个唯一的 ID，也可传递－1，这时 wxPython 将自动生成一个新的 ID，由系统来保证其唯一性。
- title——窗体的标题。
- pos——wx. Point 对象，用来指定新窗体的左上角在屏幕中的位置，通常(0，0)是显示器的左上角坐标。当将其设定为 wx. DefaultPosition，其值为(－1，－1)，表示让系统决定窗体的位置。
- size——wx. Size 对象，用来指定新窗体的初始大小。当将其设定为 wx. DefaultSize 时，其值为(－1，－1)，表示由系统来决定窗体的初始大小。
- style——指定窗体的类型的常量，wx. Frame 的常用样式如表 9-1 所示。对一个窗体控件可以同时使用多个样式，使用"位或"运算符"|"连接即可。比如 wx. DEFAULT_FRAME_STYLE 样式就是由以下几个基本样式的组合：

  ```
  wx.MAXIMIZE_BOX | wx.MINIMIZE_BOX | wx.RESIZE_BORDER | wx.SYSTEM_MENU | wx.
  CAPTION | wx.CLOSE_BOX
  ```

  要从一个组合样式中去掉个别的样式可以使用"^"按位异或操作符，例如，要创建一个默认样式的窗体，但要求用户不能缩放和改变窗体的尺寸，可以使用这样的组合：

  ```
  wx.DEFAULT_FRAME_STYLE ^ (wx.RESIZE_BORDER | wx.MAXIMIZE_BOX | wx.MINIMIZE
  _BOX)
  ```

- name——框架的名字，指定后可以使用这个名字来寻找这个窗体。

可以看到，wx. Frame. __init__()方法只有参数 parent 没有默认值，因而最简单的调用方式是

```
wx.Frame.__init__(self, parent=None)
```

这将生成一个默认位置、默认大小、默认标题的顶层窗体。

在初始化窗体时可以明确给构造函数传递一个正整数作为新窗体的 ID，此时由程序员自己来保证 ID 不重复并且没有与预定义的 ID 号冲突，例如，不能使用 wx. ID_OK(5100)、wx. ID_CANCEL(5101)、wx. ID_ANY(－1)、wx. ID_COPY(5032)、wx. ID_APPLY(5102)等预定义 ID 号对应的数值。

如果无法确定使用哪个数值作为 ID，可以使用 wx. NewId()函数来生成 ID 号，这样就可以避免确保 ID 号唯一性的麻烦。

```
id=wx.NewId()
frame=wx.Frame.__init__(None, id)
```

当然，也可以使用全局常量 wx. ID_ANY(值为－1)来让 wxPython 自动生成新的唯一 ID 号，需要时可以使用 GetId()方法来得到它，如

```
frame=wx.Frame.__init__(None, -1)
id=frame.GetId()
```

表 9-1　wx.Frame 常用样式

| 样　　式 | 说　　明 |
| --- | --- |
| wx.CAPTION | 增加标题栏 |
| wx.DEFAULT_FRAME_STYLE | 默认样式 |
| wx.CLOSE_BOX | 标题栏上显示“关闭”按钮 |
| wx.MAXIMIZE_BOX | 标题栏上显示“最大化”按钮 |
| wx.MINIMIZE_BOX | 标题栏上显示“最小化”按钮 |
| wx.RESIZE_BORDER | 边框可改变尺寸 |
| wx.SIMPLE_BORDER | 边框没有装饰 |
| wx.SYSTEM_MENU | 增加系统菜单(有“关闭”、“移动”、“改变尺寸”等功能) |
| wx.FRAME_SHAPED | 用该样式创建的框架可以使用 SetShape()方法来创建一个非矩形的窗体 |
| wx.FRAME_TOOL_WINDOW | 给框架一个比正常小的标题栏,使框架看起来像一个工具框窗体 |

本节最后通过一个具体的示例来演示使用 wxPython 创建 GUI 应用程序的思路,将下面的代码保存并运行,则会在窗体上的文本框中动态显示当前窗体的位置与大小以及鼠标相对于窗体(即窗体左上角坐标为(0,0))的当前位置,可以移动鼠标并观察值的变化。

```
import wx
class MyFrame(wx.Frame):
    def __init__(self, superior):
        wx.Frame.__init__(self, parent=superior, title=u'My First Form',
        size=(300, 300))
        self.Bind(wx.EVT_SIZE, self.OnSize)
        self.Bind(wx.EVT_MOVE, self.OnFrameMove)

        #Add a panel and some controls to display the size and position
        panel=wx.Panel(self, -1)
        label1=wx.StaticText(panel, -1, "FrameSize:")
        label2=wx.StaticText(panel, -1, "FramePos:")
        label3=wx.StaticText(parent=panel, label="MousePos:")
        self.sizeFrame=wx.TextCtrl(panel, -1, "", style=wx.TE_READONLY)
        self.posFrame=wx.TextCtrl(panel, -1, "", style=wx.TE_READONLY)
        self.posMouse=wx.TextCtrl(panel, -1, "", style=wx.TE_READONLY)
        panel.Bind(wx.EVT_MOTION, self.OnMouseMove)     #绑定事件处理函数
        self.panel=panel

        #Use some sizers for layout of the widgets
        sizer=wx.FlexGridSizer(3, 2, 5, 5)
```

```
        sizer.Add(label1)
        sizer.Add(self.sizeFrame)
        sizer.Add(label2)
        sizer.Add(self.posFrame)
        sizer.Add(label3)
        sizer.Add(self.posMouse)

        border=wx.BoxSizer()
        border.Add(sizer, 0, wx.ALL, 15)
        panel.SetSizerAndFit(border)
        self.Fit()

    def OnSize(self, event):
        size=event.GetSize()
        self.sizeFrame.SetValue("%s, %s" %(size.width, size.height))

        #tell the event system to continue looking for an event handler,
        #so the default handler will get called.
        event.Skip()

    def OnFrameMove(self, event):
        pos=event.GetPosition()
        self.posFrame.SetValue("%s, %s" %(pos.x, pos.y))

    def OnMouseMove(self, event):        #鼠标移动事件处理函数
        pos=event.GetPosition()
        self.posMouse.SetValue("%s, %s" %(pos.x, pos.y))

if __name__=='__main__':
    app=wx.App()#Create an instance of the application class
    frame=MyFrame(None)
    frame.Show(True)
    app.MainLoop() #Tell it to start processing events
```

运行结果如图9-1所示，改变窗体位置和大小时，文本框内的数值会动态变化，当鼠标在窗体内移动时，文本框内实时显示鼠标当前坐标，当鼠标移动到窗体之外时，文本框中的数值将不再变化。

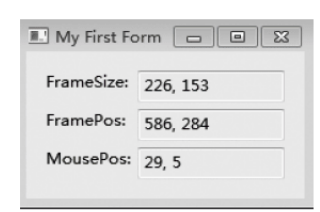



图9-1　显示窗体大小和位置以及鼠标位置

## 9.2 Controls

wxPython 提供了几乎所有常用的控件，例如按钮、静态文本标签、文本框、单选按钮、复选框、对话框、菜单、列表框、树形控件等。如需在窗体上增加其他控件，可在窗体构造函数中增加代码，如需响应和处理特定事件，可增加框架类的成员函数，并进行相应的绑定操作。本节通过几个示例来演示 wxPython 控件的用法。为了方便介绍，大致根据控件的类型将控件进行了不同的组合放在一起来介绍，而不是孤零零地逐个介绍控件的用法。

### 9.2.1 Button、StaticText、TextCtrl

按钮控件的构造函数语法如下：

```
__init__(self, Window parent, int id=-1, String label=EmptyString, Point pos
=DefaultPosition, Size size=DefaultSize, long style=0, Validator validator=
DefaultValidator, String name=ButtonNameStr)
```

按钮主要用来响应用户的单击操作，而按钮上面的文本一般是创建时直接指定的，很少需要修改。当然，如果确实需要动态修改，可以通过 SetLabelText()方法来实现这个目的，再结合 GetLabelText()方法来获取按钮控件上面显示的文本，则可以实现同一个按钮完成不同功能的目的。为按钮绑定事件处理函数的方法为：

```
Bind(event, handler, source=None, id=-1, id2=-1)
```

静态文本控件主要用来显示文本或给用户操作提示，不用来响应用户单击或双击事件，需要时可以使用 SetLabel()方法动态为 StaticText 控件设置文本。静态文本控件的构造函数语法如下：

```
__init__(self, Window parent, int id=-1, String label=EmptyString, Point pos
= DefaultPosition, Size size = DefaultSize, long style = 0, String name =
StaticTextNameStr)
```

文本框主要用来接收用户的文本输入，可以使用 GetValue()方法获取文本框中输入的内容，也使用 SetValue()方法设置文本框中的文本，文本框控件的构造函数语法如下：

```
__init__(self, Window parent, int id=-1, String value=EmptyString, Point pos
=DefaultPosition, Size size=DefaultSize, long style=0, Validator validator=
DefaultValidator, String name=TextCtrlNameStr)
```

下面通过一个示例来演示这三个控件的用法，将下面的代码保存为 wxIsPrime.py，运行后用户输入一个整数，单击按钮后判断是否为素数并输出结果。

```
import wx
from math import sqrt
```

```
class IsPrimeFrame(wx.Frame):
    def __init__(self, superion):
        wx.Frame.__init__(self, parent=superion, title='Check Prime',
        size=(400, 200))
        panel=wx.Panel(self)
        panel.SetBackgroundColour('Yellow')     #设置窗体背景颜色
        wx.StaticText(parent=panel, label='Input a integer:', pos=(10, 10))
                                                #添加静态文本控件
        self.inputN=wx.TextCtrl(parent=panel, pos=(120, 10))    #添加文本框
        self.result=wx.StaticText(parent=panel, label='', pos=(10, 50))
        self.buttonCheck=wx.Button(parent=panel, label='Check', pos=(70, 90))
                                                #添加按钮
        #为按钮绑定事件处理方法
        self.Bind(wx.EVT_BUTTON, self.OnButtonCheck, self.buttonCheck)
        self.buttonQuit=wx.Button(parent=panel, label='Quit', pos=(150, 90))
        self.Bind(wx.EVT_BUTTON, self.OnButtonQuit, self.buttonQuit)

    def OnButtonCheck(self, event):
        self.result.SetLabel('')
        try:
            num=int(self.inputN.GetValue())     #获取用户输入的数字
        except BaseException, e:
            self.result.SetLabel('not a integer')
            return
        n=int(sqrt(num))
        for i in range(2, n+1):                 #判断用户输入的数字是否为素数
            if num%i==0:
                self.result.SetLabel('No')      #使用静态文本框显示结果
                break
        else:
            self.result.SetLabel('Yes')

    def OnButtonQuit(self, event):
        dlg=wx.MessageDialog(self, 'Really Quit?', 'Caution',\
                            wx.CANCEL|wx.OK|wx.ICON_QUESTION)
        if dlg.ShowModal()==wx.ID_OK:
            self.Destroy()
if __name__=='__main__':
    app=wx.App()
    frame=IsPrimeFrame(None)
    frame.Show()
    app.MainLoop()
```

运行结果如图 9-2 所示。

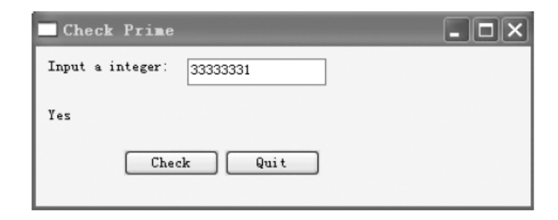



图 9-2 素数判断

## 9.2.2 Menu

菜单可以分为普通菜单和弹出式菜单两大类，其中普通菜单也就是大多数窗口菜单栏的下拉菜单，弹出式菜单也称上下文菜单，一般需要使用鼠标右键激活，并根据不同的环境或上下文来显示不同的菜单项。下面对这两类菜单分别进行介绍。

### 1. 创建普通菜单

```
self.frame=wx.Frame(parent=None, title='wxGUI', size=(640, 480))
self.panel=wx.Panel(self.frame, -1)
self.menuBar=wx.MenuBar()                          #创建菜单栏
self.menu=wx.Menu()                                #创建菜单
self.menuOpen=self.menu.Append(101, 'Open')        #创建菜单项
self.menuSave=self.menu.Append(102, 'Save')
self.menuSaveAs=self.menu.Append(103, 'Save As')
self.menu.AppendSeparator()                        #分隔符
self.menuClose=self.menu.Append(104, 'Close')
self.menuBar.Append(self.menu, '&File')            #将菜单添加至菜单栏
self.menu=wx.Menu()
self.menuCopy=self.menu.Append(201, 'Copy')
self.menuCut=self.menu.Append(202, 'Cut')
self.menuPaste=self.menu.Append(203, 'Paste')
self.menuBar.Append(self.menu, '&Edit')
```

创建菜单完成之后，通过下面的代码将创建的菜单设置为窗体菜单。

```
self.frame.SetMenuBar(self.menuBar)
```

### 2. 创建弹出式菜单

```
self.popupMenu=wx.Menu()                           #创建菜单
self.popupCopy=self.popupMenu.Append(901, 'Copy')  #创建菜单项
self.popupCut=self.popupMenu.Append(902, 'Cut')
self.popupPaste=self.popupMenu.Append(903, 'Paste')
```

接下来为窗体绑定鼠标右键单击操作：

```
self.Bind(wx.EVT_RIGHT_DOWN, self.OnRClick)
```

然后编写右键单击处理函数,用户单击鼠标右键时弹出上面定义的弹出式菜单。

```
def OnRClick(self, event):
    pos=(event.GetX(), event.GetY())                  #获取鼠标当前位置
    self.panel.PopupMenu(self.popupMenu, pos)  #在鼠标当前位置弹出上下文菜单
```

**3. 为菜单项绑定单击事件处理函数**

对于普通下拉式菜单和弹出式菜单,为菜单项绑定事件处理函数的方式是一样的,例如下面的代码,其中第二个数值型的参数是菜单项的ID,最后一个参数是事件处理函数的名称。绑定之后,运行程序并单击某菜单项,则会执行相应的事件处理函数中的代码。

```
wx.EVT_MENU(self, 102, self.OnOpen)
wx.EVT_MENU(self, 103, self.OnSave)
wx.EVT_MENU(self, 104, self.OnSaveAs)
wx.EVT_MENU(self, 105, self.OnClose)
```

**4. 编写菜单项的单击事件处理函数**

具体的事件处理函数根据不同的业务逻辑有所不同,这里仅演示如何在状态栏上显示一段文本,有关状态栏的介绍请参考9.2.3节。

```
def OnNew(self, event):
    self.statusBar.SetStatusText('You clicked the New menu.')
```

### 9.2.3 ToolBar、StatusBar

工具栏往往用来显示当前上下文最常用的功能按钮,一般而言,工具栏按钮是菜单全部功能的子集。状态栏主要用来显示当前状态或给用户友好提示,例如,Word软件中的状态栏上显示的当前页码、总页数、节数以及当前行与当前列等信息。下面分别介绍这两个控件的创建和使用方法。

**1. 创建工具栏**

```
self.toolbar=self.frame.CreateToolBar()
```

接下来在工具栏添加工具,相应的工具栏图片需要提前准备好,并存放于当前目录下。

```
self.toolbar.AddSimpleTool(9999,wx.Image('open.png',wx.BITMAP_TYPE_PNG).
ConvertToBitmap(), 'Open', 'Click to Open a file')
```

然后使用下面的代码准备工具栏使其有效。

```
self.toolbar.Realize()
```

最后绑定事件处理函数，事件处理函数的编写与前面介绍的按钮、菜单项等控件的事件处理函数一样，此处不再赘述。

```
wx.EVT_TOOL(self, 9999, self.OnOpen)
```

**2. 创建状态栏**

状态栏的创建和使用相对比较简单，通过下面的代码即可创建：

```
self.statusBar=self.frame.CreateStatusBar()
```

如果需要在状态栏上显示状态或者显示文本以提示用户，可以通过下面的代码设置状态栏文本：

```
self.statusBar.SetStatusText('You clicked the Open menu.')
```

### 9.2.4 对话框

wxPython 提供了一整套丰富的预定义对话框支持友好界面开发，常用的对话框有以下几种：

- MessageBox——简单消息框。
- GetTextFromUser——接收用户输入的文本。
- GetPasswordFromUser——接收用户输入的密码。
- GetNumberFromUser——接收用户输入的数字。
- FileDialog——文件对话框。
- FontDialog——字体对话框。
- ColourDialog——颜色对话框。

除用于信息提示的简单消息框之外，其他几种对话框的使用都遵循固定的步骤：首先创建对话框，然后显示对话框，最后根据对话框的返回值采取不同的操作。下面的代码演示了 MessageBox 用法，完整代码可以参照 9.2.5 节的示例。

```
wx.MessageBox(finalStr)
```

下面的代码演示了 MessageDialog 用法，完整代码请参考 9.2.1 节。其他对话框可以参考 MessageDialog 对话框的用法。

```
def OnButtonQuit(self, event):
    dlg=wx.MessageDialog(self, 'Really Quit?', 'Caution', wx.CANCEL|wx.OK|\
                wx.ICON_QUESTION)
    if dlg.ShowModal()==wx.ID_OK:
        self.Destroy()
```

下面的代码则在 IDLE 交互式模式下演示了颜色对话框的用法：

```
>>> import wx
>>> app=wx.App()
>>> dlg=wx.ColourDialog(None)
>>> dlg.ShowModal()
5100
>>> c=dlg.GetColourData()
>>> c
<wx._windows.ColourData; proxy of<Swig Object of type 'wxColourData *' at
0x2df84c0>>
>>> c.Colour
wx.Colour(255, 0, 0, 255)
```

### 9.2.5 RadioButton、CheckBox

单选按钮常用来实现用户在多个选项中的互斥选择,在同一组内多个选项中只能选择一个,当选择发生变化之后,之前选中的选项自动失效。单选按钮控件的构造函数语法如下:

```
__init__(self, Window parent, int id=-1, String label=EmptyString, Point pos
=DefaultPosition, Size size=DefaultSize, long style=0, Validator validator=
DefaultValidator, String name=RadioButtonNameStr)
```

可以使用 wxPython 的 SashWindow 控件对单选按钮进行分组,也可以使用单选按钮控件的样式进行分组,每组的第一个单选按钮使用 wx.RB_GROUP 样式,其他单选按钮不使用该样式。

复选框往往用来实现非互斥多选的功能,多个复选框之间的选择互不影响。复选框的构造函数语法如下:

```
__init__(self, Window parent, int id=-1, String label=EmptyString, Point pos
=DefaultPosition, Size size=DefaultSize, long style=0, Validator validator=
DefaultValidator, String name=CheckBoxNameStr)
```

单选按钮和复选框的很多操作是通用的。可以使用 GetValue()方法判断单选按钮或复选框是否被选中,使用 SetValue(True)将单选按钮或复选框设置为选中状态,使用 SetValue(False)将单选按钮或复选框设置为未选中状态。

在某些应用中,可能需要响应单选按钮、复选框的鼠标单击事件,根据不同的需要可以使用 wx.EVT_RADIOBOX()、wx.EVT_CHECKBOX()分别为单选按钮、复选框来绑定事件处理函数。

下面的代码演示了这两个控件的用法。

```
import wx
class wxGUI(wx.App):
    def OnInit(self):
```

```
        self.frame=wx.Frame(parent=None, title='wxGUI', size=(300, 280))
        self.panel=wx.Panel(self.frame, -1)

        self.radioButtonSexM=wx.RadioButton(self.panel, -1, 'Male', pos=
(80, 60))
        self.radioButtonSexF=wx.RadioButton(self.panel, -1, 'Female', pos=
(80, 80))
        self.checkBoxAdmin=wx.CheckBox(self.panel, -1, 'Aministrator', pos=
(150, 80))

        self.label1=wx.StaticText(self.panel, -1, 'UserName:', pos=(0, 110),
style=wx.ALIGN_RIGHT)
        self.label2=wx.StaticText(self.panel, -1, 'Password:', pos=(0, 130),
style=wx.ALIGN_RIGHT)

        self.textName=wx.TextCtrl(self.panel, -1, pos=(70, 110), size=(160,
20))
        self.textPwd=wx.TextCtrl(self.panel, -1, pos=(70, 130), size=(160,
20), style=wx.TE_PASSWORD)

        self.buttonOK=wx.Button(self.panel, -1, 'OK', pos=(30, 160))
        self.Bind(wx.EVT_BUTTON, self.OnButtonOK, self.buttonOK)
        self.buttonCancel=wx.Button(self.panel, -1, 'Cancel', pos=(120,
160))
        self.Bind(wx.EVT_BUTTON, self.OnButtonCancel, self.buttonCancel)
        self.buttonOK.SetDefault()

        self.frame.Show()
        return True

    def OnButtonOK(self, event):
        finalStr=''
        if self.radioButtonSexM.GetValue()==True:
            finalStr+='Sex:Male\n'
        elif self.radioButtonSexF.GetValue()==True:
            finalStr+='Sex:Female\n'
        if self.checkBoxAdmin.GetValue()==True:
            finalStr+='Administrator\n'
        if self.textName.GetValue()=='dongfuguo' and self.textPwd.
GetValue()=='dongfuguo':
            finalStr+='user name and password are correct\n'
        else:
            finalStr+='user name or password is incorrect\n'
        wx.MessageBox(finalStr)
```

```
        def OnButtonCancel(self, event):
            self.radioButtonSexM.SetValue(True)
            self.radioButtonSexF.SetValue(False)
            self.checkBoxAdmin.SetValue(True)
            self.textName.SetValue('')
            self.textPwd.SetValue('')

app=wxGUI()
app.MainLoop()
```

上面的代码运行结果如图 9-3 所示，选择单选按钮、复选框并输入文本框中要求的用户名和密码之后单击 OK 按钮会弹出消息框提示输入和选择的内容，单击 Cancel 按钮自动清除用户的输入，并默认将单选按钮 Male 设置为选中状态。

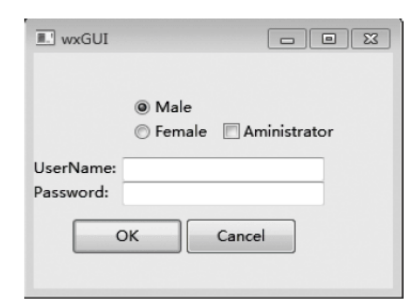



图 9-3　单选按钮、复选框的用法演示

## 9.2.6 ComboBox

组合框用来实现从固定的多个选项中选择其中一个的操作，外观与文本框类似，但是单击下拉箭头时弹出所有可选项，极大地方便了用户的操作，并且在窗体上不占用太大的空间。组合框的构造函数语法如下：

```
__init__(Window parent, int id=-1, String value=EmptyString, Point pos=
DefaultPosition, Size size=DefaultSize, List choices=EmptyList, long style=
0, Validator validator=DefaultValidator, String name=ComboBoxNameStr)
```

如果需要响应和处理组合框的鼠标单击事件，可以使用 wx.EVT_COMBOBOX()为组合框绑定事件处理函数。下面的代码演示了组合框的用法：

```
import wx
class wxGUI(wx.App):
    def OnInit(self):
        self.frame=wx.Frame(parent=None, title='wxGUI', size=(300, 200))
        self.panel=wx.Panel(self.frame, -1)

        self.names={'First Class':['Zhang San', 'Li Si', 'Wang Wu'],
                'Second Class':['Zhao Liu', 'Zhou Qi']}

        #ComboBox1
        self.comboBox1=wx.ComboBox(self.panel, value='Click here',\
                                   choices=self.names.keys(),\
                                   pos=(0, 50), size=(100, 30))
        self.Bind(wx.EVT_COMBOBOX, self.OnCombo1, self.comboBox1)
```

```
        #ComboBox2
        self.comboBox2=wx.ComboBox(self.panel, value='Click here',\
                                    choices=[],\
                                    pos=(0, 100), size=(100, 30))
        self.Bind(wx.EVT_COMBOBOX, self.OnCombo2, self.comboBox2)

        self.frame.Show()
        return True

    def OnCombo1(self, event):
        banji=self.comboBox1.GetValue()
        self.comboBox2.Set(self.names[banji]) #动态修改第二个组合框中显示的选项

    def OnCombo2(self, event):
        wx.MessageBox(self.comboBox2.GetValue())

app=wxGUI()
app.MainLoop()
```




图 9-4　组合框联动演示

程序运行后，界面如图 9-4 所示，首先在第一个组合框中选择班级，然后第二个组合框中自动列出该班级的同学姓名，选择同学姓名后弹出消息框显示选择的姓名。

### 9.2.7　ListBox

列表框用来放置多个元素提供给用户进行选择，其中每个元素都是字符串，支持用户单选和多选。列表框的样式和常用方法如表 9-2 和表 9-3 所示。

表 9-2　列表框常用样式

| 样　　式 | 说　　明 |
|---|---|
| wx. LB_EXTENDED | 可以使用 Shift 键和鼠标配合选择连续多个元素 |
| wx. LB_MULTIPLE | 可以选择多个不连续的元素 |
| wx. LB_SINGLE | 最多只能选择一个元素 |
| wx. LB_ALWAYS_SB | 始终显示一个垂直滚动条 |
| wx. LB_HSCROLL | 仅在需要时显示一个垂直滚动条 |
| wx. LB_SORT | 列表框中的元素按字母顺序排序 |

表 9-3 列表框常用方法

| 方 法 名 | 说 明 |
| --- | --- |
| Append(string) | 在列表框尾部增加一个元素 |
| Clear() | 删除列表框中所有元素 |
| Delete(index) | 删除列表框指定索引的元素 |
| FindString(string) | 返回指定元素的索引,若没找到,则返回-1 |
| GetCount() | 返回列表框中元素的个数 |
| GetSelection() | 返回当前选择项的索引,仅对单选列表框有效 |
| SetSelection(index,True/False) | 设置指定索引的元素的选择状态 |
| GetStringSelection() | 返回当前选择的元素,仅对单选列表框有效 |
| GetString(index) | 返回指定索引的元素 |
| SetString(index,string) | 设置指定索引的元素文本 |
| GetSelections() | 返回包含所选元素的元组 |
| InsertItems(items,pos) | 在指定位置之前插入元素 |
| IsSelected(index) | 返回指定索引的元素的选择状态 |
| Set(choices) | 使用列表 choices 的内容重新设置列表框 |

下面的代码演示了列表框的用法,运行程序后,列表框中显示周日到周六的每天,用户单击其中一个后弹出一个消息框来提示所选择的内容,单击 Quit 按钮时弹出关闭前的确认对话框。

```
import wx

class ListBoxDemo(wx.Frame):
    def __init__(self, superion):
        wx.Frame.__init__(self, parent=superion, title='ListBox demo', size=
        (200, 200))
        panel=wx.Panel(self)
        self.buttonQuit=wx.Button(parent=panel, label='Quit', pos=(60, 120))
        self.Bind(wx.EVT_BUTTON, self.OnButtonQuit, self.buttonQuit)
        li=['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
        'Saturday']
        self.listBox=wx.ListBox(panel, choices=li)      #创建列表框
        self.Bind(wx.EVT_LISTBOX, self.OnClick, self.listBox)
                                                        #绑定事件处理函数

    def OnClick(self, event):
        #t=self.listBox.GetSelection()
        #s=self.listBox.GetString(t)
```

```
            s=self.listBox.GetStringSelection()
            wx.MessageBox(s)

        def OnButtonQuit(self, event):
            dlg=wx.MessageDialog(self, 'Really Quit?', 'Caution',\
                                  wx.CANCEL|wx.OK|wx.ICON_QUESTION)
            if dlg.ShowModal()==wx.ID_OK:
                self.Destroy()
    if __name__=='__main__':
        app=wx.App()
        frame=ListBoxDemo(None)
        frame.Show()
        app.MainLoop()
```

运行结果如图 9-5 所示。

图 9-5 列表框用法

## 9.2.8 TreeCtrl

树形控件常用来显示有严格层次关系的数据，可以非常清晰地表示各元素之间的从属关系或层级关系，比如 Windows 资源管理器左侧窗口（见图 9-6）以及注册表编辑器（见图 9-7）。

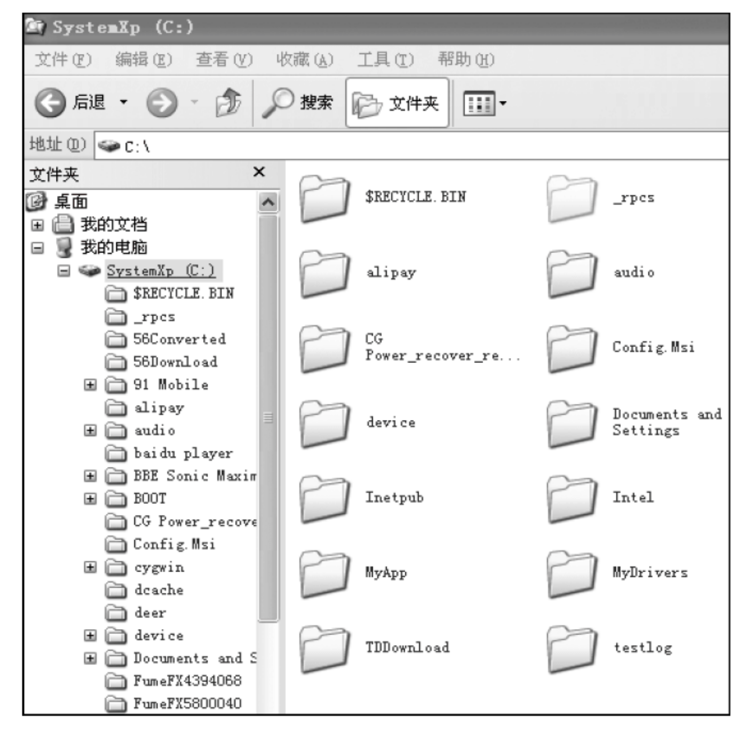



图 9-6 资源管理器界面

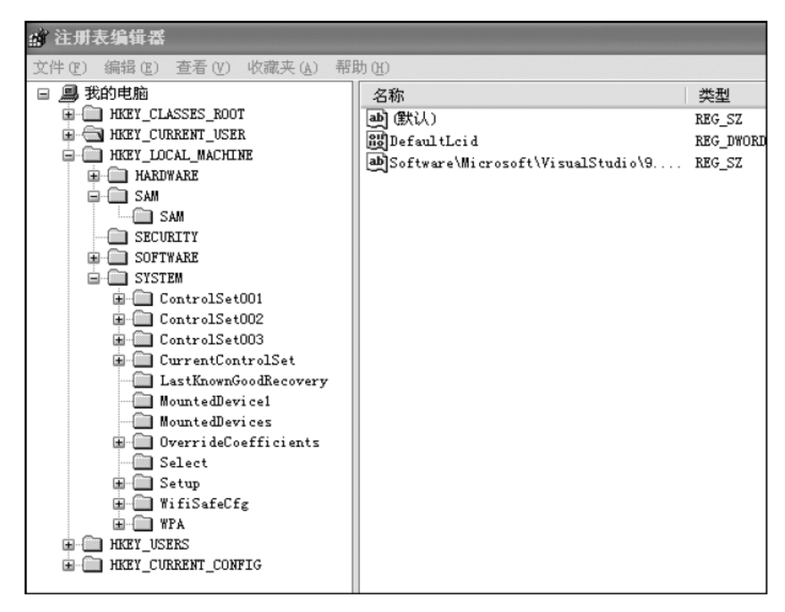



图 9-7 注册表编辑器界面

树形控件的构造函数语法如下：

```
__init__(self, Window parent, int id=-1, Point pos=DefaultPosition, Size size
= DefaultSize, long style = TR _ DEFAULT _ STYLE, Validator validator =
DefaultValidator, String name=TreeCtrlNameStr)
```

由于构造函数中大多数参数均有默认值，使用下面的代码就可以创建一个简单的树形控件：

```
tree=wx.TreeCtrl(panel)
```

树形控件的常用方法和事件分别如表 9-4 和表 9-5 所示。

**表 9-4 树形控件常用方法**

| 方　法 | 说　明 |
|---|---|
| root＝tree.AddRoot(string) | 增加根节点，返回根节点 ID |
| child＝tree.AppendItem(item，string) | 为指定节点增加下级节点，返回新节点 ID |
| SetItemText(item，string) | 设置节点文本 |
| GetItemText() | 返回节点文本 |
| SetItemPyData(item，ojb) | 设置节点数据 |
| GetItemPyData(item) | 返回指定节点的数据 |
| Expand(item) | 展开指定节点，但不展开下级节点 |
| ExpandAll() | 展开所有节点 |

续表

| 方　　法 | 说　　明 |
| --- | --- |
| Collapse(item) | 收起指定节点 |
| CollapseAndReset() | 收起指定节点并删除其下级节点 |
| GetRootItem() | 返回根节点 ID |
| (childID，cookie)＝GetFirstChild(item) | 返回指定节点的第一个子节点 |
| flag＝child. IsOk() | 测试节点 ID 是否有效 |
| (item，cookie)＝GetNextChild(item，cookie) | 返回同级的下一个节点 |
| GetLastChild(item) | 返回指定节点的最后一个子节点 |
| GetPrevSibling(item) | 返回同级的上一个节点 |
| GetItemParent(item) | 返回指定节点的父节点 ID |
| ItemHasChildren(item) | 测试节点是否有下级节点 |
| SetItemHasChildren(item，True) | 将指定节点设置为有下级节点的状态 |
| GetSelection() | 返回单选树中当前被选中节点的 ID |
| GetSelections() | 返回多选树中所有被选中节点 ID 的列表 |
| SelectItem(item，True/False) | 改变节点的选择状态 |
| IsSelected(item) | 测试节点是否被选中 |
| Delete(item) | 删除指定 ID 的节点 |
| DeleteAllItems() | 删除所有节点 |
| DeleteChildren(item) | 删除指定 ID 的节点所有下级节点 |
| InsertItem(parent，idPrevious，text) | 在指定节点后面插入节点 |
| InsertItemBefore(parent，index，text) | 在指定位置之前插入节点 |

**表 9-5　树形控件常用事件**

| 事　　件 | 说　　明 |
| --- | --- |
| wx. EVT_TREE_SEL_CHANGING | 控件发生选择变化之前触发该事件 |
| wx. EVT_TREE_SEL_CHANGED | 控件发生选择变化之后触发该事件 |
| wx. EVT_TREE_ITEM_COLLAPSING | 收起一个节点之前触发该事件 |
| wx. EVT_TREE_ITEM_COLLAPSED | 收起一个节点之后触发该事件 |
| wx. EVT_TREE_ITEM_EXPANDING | 展开一个节点之前触发该事件 |
| wx. EVT_TREE_ITEM_EXPANDED | 展开一个节点之后触发该事件 |

下面的代码演示了树形控件的用法，这个简单的示例演示了增加根节点、增加子节点、删除节点等功能。

```
import wx

class TreeCtrlFrame(wx.Frame):
    def __init__(self, superion):
        wx.Frame.__init__(self, parent=superion, title='TreeCtrl demo',
        size=(300, 400))
        panel=wx.Panel(self)
        self.tree=wx.TreeCtrl(parent=panel, pos=(5, 5), size=(120, 200))
        self.inputString=wx.TextCtrl(parent=panel, pos=(150, 10))
        self.buttonAddChild=wx.Button(parent=panel, label='AddChild', pos=
        (150, 90))
        self.Bind(wx.EVT_BUTTON, self.OnButtonAddChild, self.buttonAddChild)
        self.buttonDeleteNode=wx.Button(parent=panel, label='DeleteNode',
        pos=(150, 120))
        self.Bind(wx.EVT_BUTTON, self.OnButtonDeleteNode, self.buttonDeleteNode)
        self.buttonAddRoot=wx.Button(parent=panel, label='AddRoot', pos=
        (150, 150))
        self.Bind(wx.EVT_BUTTON, self.OnButtonAddRoot, self.buttonAddRoot)

    def OnButtonAddChild(self, event):
        itemSelected=self.tree.GetSelection()
        if not itemSelected:
            wx.MessageBox('Select a Node first.')
            return
        itemString=self.inputString.GetValue()
        self.tree.AppendItem(itemSelected, itemString)

    def OnButtonDeleteNode(self, event):
        itemSelected=self.tree.GetSelection()
        if not itemSelected:
            wx.MessageBox('Select a Node first.')
            return
        self.tree.Delete(itemSelected)

    def OnButtonAddRoot(self, event):
        rootItem=self.tree.GetRootItem()
        if rootItem:
            wx.MessageBox('The tree has already a root.')
        else:
            itemString=self.inputString.GetValue()
            self.tree.AddRoot(itemString)

if __name__=='__main__':
    app=wx.App()
```

```
frame=TreeCtrlFrame(None)
frame.Show()
app.MainLoop()
```

运行结果如图 9-8 所示，界面中的文本框用来输入节点显示的文本，首先单击 AddRoot 按钮插入根节点，然后单击选中根节点，再单击 AddChild 按钮插入子节点，最后单击选中某个子节点再单击 AddChild 按钮为其插入子节点。如果单击选中某个子节点后单击 DeleteNode 按钮则可以删除该节点。

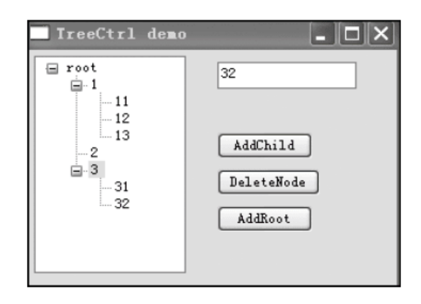



图 9-8　树形控件用法

## 9.3　Boa-constructor

Boa-constructor 是基于 wxPython 开发的一款优秀的界面设计软件，同时也是一款不错的 Python 集成开发环境，使用 Boa-constructor 设计界面时可通过鼠标调整控件的大小和位置，而不用像 IDLE 那样完全依靠手写代码来创建和控制界面。

登录 http://sourceforge.net/projects/boa-constructor 下载 Boa-constructor 安装程序后，安装过程如同其他 Windows 应用程序一样，无须多做解释。启动 Boa-constructor 时需要首先打开 IDLE，然后在 IDLE 中打开并运行 C:\Python27\Lib\site-packages\boa-constructor\Boa.py 文件，或者直接双击运行该文件，当然也可以在桌面或者更习惯的位置创建快捷方式。启动之后的界面基本上一目了然，本书不再详述，请大家自行查阅相关资料。

## 本章小结

（1）wxPython 是跨平台的 GUI 模块，除此之外，还可以使用 Python 内置的 Tkinter 以及基于 Java 的 Jython 和支持.NET 的 IronPython 等开发环境来支持 GUI 编程。

（2）框架类 Frame 是可以包含标题栏、菜单、按钮、单选按钮、文本框、组合框等其他控件的容器。

（3）如果无法确定使用哪个数值作为控件 ID，可以使用 wx.NewId()函数来生成 ID 号，这样就可以避免确保 ID 号唯一性的麻烦。也可以使用全局常量 wx.ID_ANY（值为 −1）来让 wxPython 自动生成新的唯一 ID 号，需要时可以使用 GetId()方法来得到它。

(4) 静态文本控件 StaticText 一般不用来响应用户的鼠标单击、双击等交互操作。

(5) 文本框主要用来接收用户的文本输入,可以使用 GetValue()方法获取文本框中输入的内容,也使用 SetValue()方法设置文本框中的文本。

(6) 按钮控件 Button 上显示的文本可以通过 SetLabelText()方法动态改变,结合获取文本的 GetLabelText()方法可以让一个按钮实现多个功能。

(7) 下拉菜单和弹出式菜单的菜单项事件处理函数绑定方式是一样的。

(8) 对话框需要首先创建,然后运行并显示对话框,最后再获取对话框的返回值。

(9) 同一组中的多个单选按钮控件的选择是互斥的,而复选框控件的选择不是互斥的。

(10) 使用组合框控件的 Set()方法可以改变组合框的可选项列表。

(11) 树形控件常用来显示有严格层次关系或从属关系的数据。

## 习题

9.1 设计一个窗体,并放置一个按钮,单击按钮后弹出颜色对话框,关闭颜色对话框后提示选中的颜色。

9.2 设计一个窗体,并放置一个按钮,按钮默认文本为"开始",单击按钮后文本变为"结束",再次单击后变为"开始",循环切换。

9.3 设计一个窗体,模拟 QQ 登录界面,当用户输入号码 123456 和密码 654321 时提示正确,否则提示错误。

# 附录 A　将 Python 程序转换为 exe 程序

将 Python 程序转换为 exe 版本可执行程序之后再发布，可以在没有安装 Python 环境的 Windows 平台上运行，这个功能极大地方便了用户。为了将 Python 程序转换为 exe 可执行文件，需要用到 py2exe 和 distutils 模块。当然，首先应保证所编写的 Python 程序可以正常运行，并且本机已安装了所有需要的扩展模块和相关的动态链接库文件。

例如，假设有 Python 源程序文件 CheckAndViewAutoRunsInSystem.py，然后编写 setup.py 文件，内容为：

```
import distutils
import py2exe
distutils.core.setup(console=['CheckAndViewAutoRunsInSystem.py'])
```

然后在命令提示符下执行下面的命令：

```
python setup.py py2exe
```

接下来就会看到控制台窗口中大量的提示内容飞快地闪过，这个过程将自动搜集 CheckAndViewAutoRunsInSystem.py 程序执行所需要的所有支持文件，如果创建成功的话则会在当前文件夹下生成一个 dist 子文件夹，其中包含了最终程序执行所需要的所有内容。等待编译完成以后，将 dist 文件中的文件打包发布即可。例如，上面的步骤完成之后，dist 文件夹中文件列表如图附 A-1 所示。




图附 A-1　dist 文件夹文件列表

py2exe 模块的详细用法可以查阅有关资料，但是对于一般应用而言，上面的代码已经足够了。唯一要注意的问题是，对于控制台应用程序，要想转换为 exe 可执行程序直接套用上面的代码框架即可，仅需要把

```
distutils.core.setup(console=['CheckAndViewAutoRunsInSystem.py'])
```

这行代码中的文件名替换为自己的 Python 程序文件名即可。而对于 GUI 应用程序，还需要将上面代码中的关键字 console 修改为 windows。

# 附录 B　常用 Python 扩展库简介

## B.1　图形图像编程模块

Python 的跨平台扩展模块 PyOpenGL 封装了 OpenGL API，支持图形编程所需要的所有功能，包括绘制二维和三维图形、绘制文本、闭合区域填充、图形几何变换、纹理映射、光照计算以及响应和处理键盘与鼠标事件等。使用下面的方式导入该模块之后，即可进行图形编程。

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
```

Python Imaging Library (PIL)是支持 Python 的图像处理扩展模块，支持多种图像格式，并提供了非常强大的图像处理功能，包括查看图像灰度直方图、图像点运算、缩放、旋转、裁剪、增强、边缘提取等。PIL 模块需要单独进行安装后才能使用，在 PIL 中主要提供了 Image、ImageChops、ImageColor、ImageDraw、ImagePath、ImageFile、ImageEnhance、PSDraw 以及其他一些模块来支持图像的处理。

## B.2　游戏编程模块

Pygame 是一套在优秀 SDL 库基础上开发的免费的 Python 游戏编程和多媒体应用开发模块，该模块易学易用且高度可移植，使用了 OpenGL、DirectX、windib 等多个后端来保证游戏运行稳定性，核心功能使用优化的 C 语言代码和汇编代码编写来提高运行速度，并且可以更容易地使用多核 CPU，可以运行于几乎所有平台和操作系统。Pygame 模块包含了大量其他支持游戏和多媒体编程的模块，如表附 B-1 所示。

**表附 B-1　pygame 主要模块**

| 模　块 | 说　　明 | 模　块 | 说　　明 |
|---|---|---|---|
| display | 屏幕显示 | surface | 绘制屏幕 |
| event | 事件处理 | time | 时间控制 |
| image | 图像处理 | cursors | 控制鼠标指针 |
| mixer | 音乐编程 | transform | 修改和移动图像 |
| mouse | 鼠标消息处理 | key | 读取键盘按键 |
| movie | 视频文件播放，需要安装 PyMedia | font | 使用字体 |

## B.3 语音识别模块

在 Windows 平台上使用 Python 编写语音识别程序需要用到 speech 模块，并且需要安装 Pywin32 和 Microsoft Speech SDK。speech 模块支持的主要功能有：文本合成语音，将键盘输入的文本信息转换为语音信号方式输出；语音识别，将输入的语音信号识别为文本；特定词的识别，对输入的语音信号进行特定词的捕捉；特定用户、特定词的识别，能够对不同人、不同特定词进行识别。

speech 模块的主要方法如表附 B-2 所示。

**表附 B-2 speech. py 模块主要方法**

| 方　　法 | 说　　明 |
| --- | --- |
| speech. say(phrase) | 读出给定的文本 |
| speech. input(prompt=None，phraselist=None) | 打印信息 prompt 提示用户使用语音录入在 phraselist 中列出的文本，并返回用户录入的内容。该方法会阻塞当前线程直至得到用户录入或者按 Ctrl+C 键结束 |
| speech. listenfor(phraselist，callback) | 如果用户语音录入 phraselist 中的任何文本，则自动调用回调函数 callback，并返回 Listener 对象 |
| speech. listenforanything(callback) | 得到用户语音录入的内容后自动执行回调函数 callback(spoken_text，listener)，并返回 Listener 对象 |
| speech. Listener. islistening(self) | 当 Listener 对象处于监听状态时返回 True |
| speech. Listener. stoplistening(self) | 停止监听，当 Listener 对象处于监听状态时返回 True |
| speech. islistening() | 只要有 Listener 对象正在监听则返回 True |
| speech. stoplistening() | 停止所有 Listener 对象的监听状态，如果有 Listener 对象处于监听状态则返回 True |

## B.4 网络编程模块

Python 提供了 socket 模块，对 Socket 进行了二次封装，支持 Socket 接口的访问，大幅度简化了程序的开发流程，提高了开发效率。除 socket 模块之外，Python 还提供了 urllib、httplib 等大量模块可以对网页内容进行读取和处理，在此基础上结合多线程编程以及其他有关模块可以快速开发网页爬虫之类的应用。最后，可以使用 Python 语言编写 CGI 程序，也可以把 Python 代码嵌入到网页中运行，而借助于 web2py 框架，则可以快速开发网站应用。

## B.5 多线程编程模块

threading 模块是 Python 支持多线程编程的重要模块，该模块是在底层模块“_thread”的基础上开发的更高层次的线程编程接口，提供了大量的方法和类来支持多线程编程，极大地方便了用户。threading 模块提供了 Thread、Lock、RLock、Condition、Event、Timer、Semaphore 等大量类来支持多线程编程，Thread 是其中最重要也是最基本的一个类，可以通过该类创建线程并控制线程的运行，而 Lock、RLock、Condition、Semaphore 等类则用来支持线程同步。

## B.6 数据库编程模块

可以通过 Python 标准模块 sqlite3 访问 SQLite 数据库。SQLite 是内嵌在 Python 中的轻量级、基于磁盘文件的数据库管理系统，不需要服务器进程，支持使用 SQL 语句来访问数据库。该数据库使用 C 语言开发，支持大多数 SQL91 标准，支持原子的、一致的、独立的和持久的事务，但不支持外键限制；通过数据库级的独占性和共享锁定来实现独立事务，当多个线程同时访问同一个数据库并试图写入数据时，每一时刻只有一个线程可以真正地写入数据。

SQLite 支持 2TB 大小的单个数据库，每个数据库完全存储在单个磁盘文件中，以 B+树数据结构的形式存储，一个数据库就是一个文件，通过简单的文件复制即可实现数据库的备份。如果需要使用可视化管理工具，可以下载并使用 SQLiteManager、SQLite Database Browser 或其他类似工具。

另外，可以使用 Pywin32 模块来访问 Access 数据库，使用 Pywin32 或 pymssql 两种不同的方式来访问 MS SQL Server 数据库，可以使用 MySQLDb 模块访问 MySQL 数据库。

## B.7 Pywin32

Pywin32 即 Python for Windows Extensions，提供了 Python 访问和调用 Windows 底层功能函数的接口，Pywin32 包括了 win32api、win32com、win32gui、win32process 等模块。下面的代码演示了如何使用 Pywin32 来检查随计算机启动而自动启动的程序列表：

```
from win32api import *
from win32con import *

def GetValues(fullname):
    name=str.split(fullname, '\\', 1)
    try:
```

```
            if name[0]=='HKEY_LOCAL_MACHINE':
                key=RegOpenKey(HKEY_LOCAL_MACHINE, name[1], 0, KEY_READ)
            elif name[0]=='HKEY_CURRENT_USER':
                key=RegOpenKey(HKEY_CURRENT_USER, name[1], 0, KEY_READ)
            elif name[0]=='HKEY_CURRENT_ROOT':
                key=RegOpenKey(HKEY_CURRENT_ROOT, name[1], 0, KEY_READ)
            elif name[0]=='HKEY_CURRENT_CONFIG':
                key=RegOpenKey(HKEY_CURRENT_CONFIG, name[1], 0, KEY_READ)
            elif name[0]=='HKEY_USERS':
                key=RegOpenKey(HKEY_USERS, name[1], 0, KEY_READ)
            else:
                print 'Error, no key named ', name[0]
            info=RegQueryInfoKey(key)
            for i in range(0, info[1]):
                ValueName=RegEnumValue(key, i)
                print str.ljust(ValueName[0], 20), ValueName[1]
            RegCloseKey(key)
        except BaseException, e:
            print 'Sth is wrong'
            print e
if __name__=='__main__':
     KeyNames = ['HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
CurrentVersion\\Run',
                  'HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
CurrentVersion\\RunOnce',
                  HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
CurrentVersion\\RunOnceEx',
                  'HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\
CurrentVersion\\Run',
                  'HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\
CurrentVersion\\RunOnce']
    for KeyName in KeyNames:
        print KeyName
        GetValues(KeyName)
```

## B.8 ctypes

ctypes 是 Python 用来处理动态链接库的标准扩展模块，提供了与 C 语言兼容的数据类型，允许在 Python 程序中调用动态链接库或共享库中的代码，从而支持 Python 与其他编程语言的混合编程，充分发挥各自的优势，大幅度提高开发效率和运行效率。

ctypes 提供了三种方法调用动态链接库的函数：cdll、windll 和 oledll，它们的不同之处在于函数调用时的参数传递方式和返回时栈的平衡方式。cdll 加载的动态链接库导出

的函数必须使用标准的 cdecl 调用约定(函数的参数从右往左依次压入栈内,在函数执行完成后,由函数的调用者负责函数的栈帧平衡),windll 方法加载的动态链接库导出的函数必须使用 stdcall 调用约定(Win32 API 的原生约定),oledll 方法和 windll 类似,不过假设函数返回一个 HRESULT 错误代码。

下面的代码调用 Windows 动态链接库 user32. dll 中的 MessageBoxA()函数来显示对话框。

```
>>> import ctypes          #通过 ctypes 可以调用动态链接库中的函数
>>> user32=ctypes.windll.LoadLibrary('user32.dll')
>>> user32.MessageBoxA(0, str.encode('Hello world!'), str.encode('Python
ctypes'), 0)
1
```

或者使用下面更为简洁的形式:

```
>>> import ctypes
>>> ctypes.windll.user32.MessageBoxA(0, str.encode('Hello world!'), str.
encode('Python ctypes'), 0)
```

下面的代码调用标准 C 语言函数库 msvcrt 中的 printf()函数来输出文本。

```
import ctypes
msvcrt=ctypes.cdll.LoadLibrary('msvcrt')
printf=msvcrt.printf
printf('Hello world!')
```

或者使用下面形式:

```
import ctypes
ctypes.cdll.msvcrt.printf('Hello world!')
```

该程序需要在命令提示符环境中而不是在 IDLE 中执行,如果在 IDLE 环境中运行输出的是字符数量而不是字符,例如,将上面的代码复制到 IDLE 交互窗口执行结果如下:

```
>>> import ctypes
>>> ctypes.cdll.msvcrt.printf('Hello world!')
12
```

## B.9 科学计算与可视化模块

用于科学计算与可视化的 Python 模块非常多,例如 NumPy、SciPy、SymPy、Matplotlib、Traits、TraitsUI、Chaco、TVTK、Mayavi、VPython、OpenCV。其中,NumPy 模块是科学计算包,提供了 Python 中没有的数组对象,支持 N 维数组运算、处理大型矩阵、成熟的广播函数库、矢量运算、线性代数、傅里叶变换以及随机数生成等功能,并可与

C++、FORTRAN 等语言无缝结合。SciPy 模块依赖于 NumPy，提供了更多的数学工具，包括矩阵运算、线性方程组求解、积分、优化等。Matplotlib 模块依赖于 NumPy 模块和 tkinter 模块，可以绘制多种形式的图形，包括线图、直方图、饼状图、散点图、误差线图等，是计算结果可视化的重要工具。其中 SciPy 主要模块如表附 B-3 所示。

**表附 B-3 SciPy 主要模块**

| 模 块 | 说 明 |
|---|---|
| constants | 常数 |
| special | 特殊函数 |
| optimize | 数值优化算法，如最小二乘拟合(leastsq)、函数最小值(fmin 系列)、非线性方程组求解(fsolve)等等 |
| interpolate | 插值(interp1d、interp2d 等等) |
| integrate | 数值积分 |
| signal | 信号处理 |
| ndimage | 图像处理，包括 filters 滤波器模块、fourier 傅里叶变换模块、interpolation 图像插值模块、measurements 图像测量模块、morphology 形态学图像处理模块等等 |
| stats | 统计 |

## B.10 软件分析插件

IDAPython 是运行于交互式反汇编器 IDA 的插件，用于实现 IDA 的 Python 编程接口。IDA 在逆向工程领域具有广泛的应用，尤其是二进制文件静态分析，其强大的反汇编功能一直处于业内领先水平。IDAPython 插件使得 Python 脚本程序能够在 IDA 中运行并实现自定义的软件分析功能，通过该插件运行的 Python 脚本程序可以访问整个 IDA 数据库，并且可以方便地调用所有 IDC 函数和使用所有已安装的 Python 模块中的功能。目前 IDAPython 还不支持 Python 3.x，较高版本的 IDA 中集成了 IDAPython 插件，如果需要安装或升级，需要登录其官方网站下载安装适合当前已安装 Python 和 IDA 版本的 IDAPython 插件。

## B.11 其他常用模块

除了前面介绍的 Python 模块，下面简要介绍另外一些常用的模块。

- cookielib：提供了可存储和操作 cookie 的对象与方法，以便于与 urllib 模块配合使用来访问 Internet 资源。
- httplib：是相对底层的 http 请求处理模块，不如 urllib 模块封装的层次高，但能够更加灵活地对通信进行控制。
- beautifulSoup：使用 Python 实现的一个 HTML/XML 的解析器，可以很好地处

理不规范标记并生成剖析树(parse tree)。

- Scrapy：使用 Python 实现的一个快速、高层的屏幕抓取和 Web 抓取框架，用于抓取 Web 站点并从页面中提取结构化的数据。Scrapy 用途非常广泛，可以用于数据挖掘、监测和自动化测试等。
- json、BeJson：轻量级数据交换格式，易于阅读和编写，同时也易于机器解析和生成，使用该格式可以提高网络传输速度。
- rsa：密码模块，可以用来结合 urllib 模块实现某些网站或论坛的模拟登录。
- sys：提供了与解释器使用和维护有关对象的接口。
- math：提供了常用的数学函数。
- Locale：提供了 C 语言本地化函数的接口，并提供相关函数实现基于当前 locale 设置的数字与字符串转换。
- random：提供了随机数生成函数以及随机化有关的函数。
- datetime：支持日期时间有关功能。
- urllib/urllib2：网页读取与访问。
- fabric：远程操作与部署。
- capstone：反汇编框架。
- ropper：ROP 相关框架。
- Yara：恶意软件识别与分类引擎。
- audioop：用于处理原始音频数据。
- codecs：数据和流的编码与解码。
- difflib：用于计算对象的区别。
- email：支持电子邮件数据的解析、处理与生成。
- xml：XML 处理模块。

# 附录C　安卓平台的Python编程

Python也可以在运行安卓系统的手机或其他移动终端上安装并进行编程，目前常用的开发环境有QPython(支持Python 2.7.2)、QPython3(支持Python 3.2.2)或Compiler(支持包括Python在内的多种脚本语言)，都可以通过手机助手来安装。这里以QPython来介绍安卓平台的Python编程。安装QPython成功以后，在手机桌面上可以看到QPython主程序，运行该程序后主界面如图附C-1所示。




图附C-1　QPython主界面

在主界面上可以通过不同的菜单来进行终端交互式开发模式或程序开发模式，例如单击“终端”进行交互式开发模式，图附C-2和图附C-3分别演示了列表推导式以及random模块的math模块的用法。

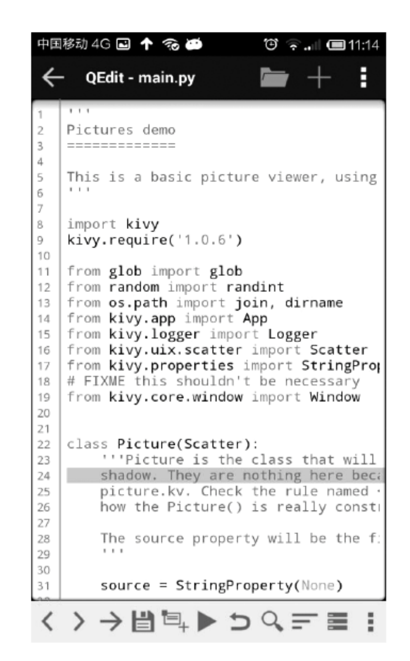


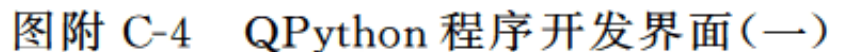

图附C-2　random模块以及列表推导式




图附C-3　基本表达式和math模块用法

图附C-4和图附C-5演示了程序开发模式的用法。当然了，在手机上编写程序还是非常不方便的，可以在计算机上编写好程序后再复制到手机上运行，具体的编程过程与本书介绍的基本一致，有关的扩展库知识和详细的安卓开发可以查阅更多资料。



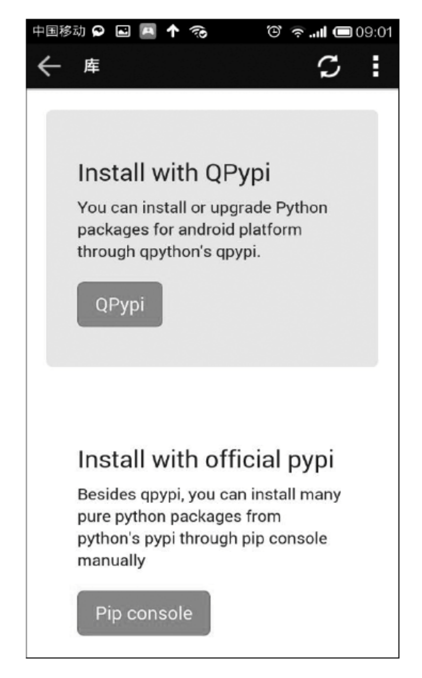


图附 C-4　QPython 程序开发界面(一)

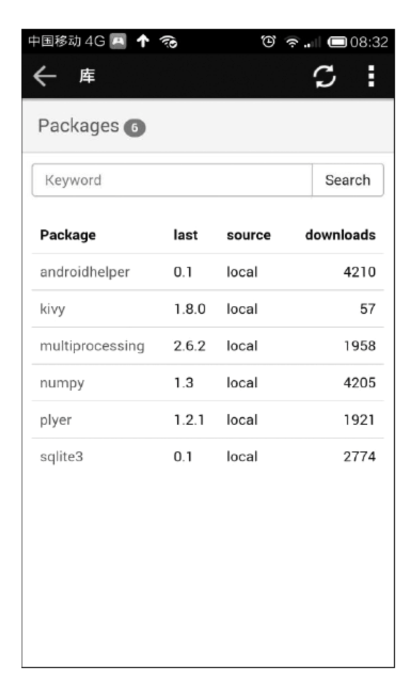



图附 C-5　QPython 程序开发界面(二)

在安卓手机平台上安装 QPython 之后，可以使用 QPypi 或 pip 工具来管理 Python 扩展库，QPython 扩展库管理主界面和 QPypi 界面分别如图附 C-6 和图附 C-7 所示。pip 工具支持的命令主要有：




图附 C-6　QPython 扩展库管理主界面




图附 C-7　QPypi 界面

（1）bundle，创建包含多个包的 pybundles；
（2）freeze，显示所有已安装的包；
（3）help，显示可用命令；
（4）install，安装包；
（5）search，搜索 PyPi；
（6）uninstall，卸载包；
（7）unzip，解压缩单个包；
（8）zip，压缩单个包。

# 参 考 文 献


[1] www. python. org.

[2] Python 3.4 Manuals.

[3] Python 2.7 Manuals.

[4] 董付国. Python 程序设计. 北京：清华大学出版社，2015.

[5] 张颖，赖勇浩. 编写高质量代码——改善 Python 程序的 91 个建议. 北京：机械工业出版社，2014.

[6] 杨佩璐，宋强，等. Python 宝典. 北京：电子工业出版社，2014.

[7] Toby Donaldson. Python 编程入门. 袁国忠译. 北京：人民邮电出版社，2013.

[8] 赵家刚，狄光智，吕丹桔，等. 计算机编程导论——Python 程序设计. 北京：人民邮电出版社，2013.